

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

**A TOOL FOR EFFICIENT EXECUTION AND
DEVELOPMENT OF REPETITIVE TASK GRAPHS
ON A DISTRIBUTED MEMORY MULTIPROCESSOR**

by

Charles Brian Koman

September 1995

Thesis Advisor:

Amr Zaky

Approved for public release; distribution is unlimited.

19960401 032

DTIC QUALITY INSPECTED 1

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE September 1995		3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE A TOOL FOR EFFICIENT EXECUTION AND DEVELOPMENT OF REPETITIVE TASK GRAPHS ON A DISTRIBUTED MEMORY MULTIPROCESSOR (U)				5. FUNDING NUMBERS	
6. AUTHOR(S) Koman, Charles Brian					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSORING/ MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.					
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The major problem addressed by this research is the development of one or more scheduling heuristics suitable for applications which involve repetitive execution of task graphs on a distributed memory multiprocessor, and to test the performance of these heuristics on a multiprocessor. The approach taken was to create more than one modified version of the PS heuristic previously introduced [KAS 94]. The modifications aim to provide a more realistic characterization of the computation-communication mechanism for the machine used in the experiments. In order to identify these characteristics, the performance of the system was comprehensively tested using different kinds of experiments. In addition, tools were developed to facilitate the development of acyclic applications. The programming tools developed require the programmer to write the program such that each node of the graph is a separate function. These functions are then packaged and converted to compilable source code in a high level programming language. The heuristic were tested using two actual applications, the correlator and Gaussian elimination, and a set of randomly created acyclic task graphs whose structure resembles realistic applications. These task graphs were created, scheduled, and packaged using RPS. Task graphs scheduled using RPS are shown to produce, on the average, efficiencies of 67 percent on four processors and 59 percent with eight processors for graphs with a 10 to 1 computation-communication ratio. The other extreme, graphs with a 1 to 1 computation-communication ratio, produced no appreciable speedup.					
14. SUBJECT TERMS Distributed Memory, Throughput, Periodic Scheduling				15. NUMBER OF PAGES 157	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL		

Approved for public release; distribution is unlimited

**A TOOL FOR EFFICIENT EXECUTION AND DEVELOPMENT
OF REPETITIVE TASK GRAPHS ON A DISTRIBUTED
MEMORY MULTIPROCESSOR**

Charles Brian Koman
Lieutenant, United States Navy
B.E.E., Georgia Institute of Technology, 1986

Submitted in partial fulfillment of the
requirements for the degree of


MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

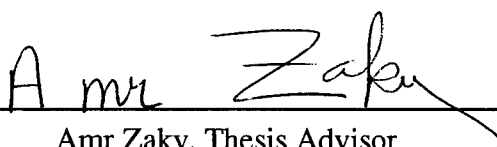
NAVAL POSTGRADUATE SCHOOL

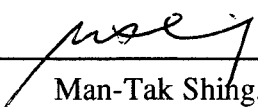
September 1995


Author:


Charles Brian Koman

Approved by:


Amr Zaky, Thesis Advisor


Man-Tak Shing, Second Reader


Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The major problem addressed by this research is the development of one or more scheduling heuristics suitable for applications which involve repetitive execution of task graphs on a distributed memory multiprocessor, and to test the performance of these heuristics on a multiprocessor.

The approach taken was to create more than one modified version of the PS heuristic previously introduced [KAS 94]. The modifications aim to provide a more realistic characterization of the computation-communication mechanism for the machine used in the experiments. In order to identify these characteristics, the performance of the system was comprehensively tested using different kinds of experiments. In addition, tools were developed to facilitate the development of acyclic applications. The programming tools developed require the programmer to write the program such that each node of the graph is a separate function. These functions are then packaged and converted to compilable source code in a high level programming language.

The heuristic were tested using two actual applications, the correlator and Gaussian elimination, and a set of randomly created acyclic task graphs whose structure resembles realistic applications. These task graphs were created, scheduled, and packaged using RPS. Task graphs scheduled using RPS are shown to produce, on the average, efficiencies of 67 percent on four processors and 59 percent with eight processors for graphs with a 10 to 1 computation-communication ratio. The other extreme, graphs with a 1 to 1 computation-communication ratio, produced no appreciable speedup.

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	OBJECTIVES	1
1.	Scope of the Thesis	3
B.	THESIS ORGANIZATION	3
II.	BACKGROUND	5
A.	THE MAPPING PROBLEM	5
B.	PRIOR WORK	6
1.	Graph Partitioning	7
2.	Mapping	7
3.	Summary	11
C.	PS HEURISTIC	11
1.	Processor Selection	13
2.	Instance Overlap	14
3.	Example	15
III.	HARDWARE AND SOFTWARE CHARACTERISTICS	17
A.	INMOS TRANSPUTERS	17
1.	General Information	17
2.	IMS T800	18
3.	IMS B003	18
4.	IMS B004	22
B.	PARASOFT EXPRESS	26
1.	System Configuration	26
2.	Communications Functions	27
a.	Processor Allocation	28
b.	Loading Programs	28
c.	Message Passing	28
d.	Additional Communications Functions	31
e.	Multitasking	32
3.	Familiarization Testing	32
a.	Simple Message Passing	32
b.	Changing Message Types	33
c.	Sending and Receiving Multiple Messages	33
d.	Sending Messages Internally	33
e.	Non-blocking Reads	34
f.	Multitasking	34
g.	Program Loading	34
h.	Program Looping	35
i.	Message Reception Order	35
4.	Communications Model	35
a.	Message Passing Testing	36
b.	Internal Message Passing	43

	c.Routing Delays	44
IV.	RPS HEURISTIC AND PROGRAMMING TOOLS	49
A.	RPS SCHEDULER	49
1.	Transputer/Express System Model	49
a.	The Communications Model	50
b.	Routing Overhead Model	50
c.	Determining Routing Paths	51
d.	Input and Output Effects	51
2.	Input to the Scheduler	52
3.	RPS Details	54
a.	Scheduling Order	54
b.	Processor Assignment	54
c.	Index Assignment	59
B.	RPS PACKAGER	61
1.	Read and Write Commands	61
2.	Code Generation Information Files	62
3.	Generated Node Code	63
a.	Flow of Execution	64
b.	Node Requirements	66
c.	Node .tcs Files	67
d.	Same Processor Buffering	68
e.	Node Indexing Effects	68
4.	Host PC Code	69
a.	Execution Flow	70
b.	Message Passing and I/O for the Host	73
c.	Effect of Indices on I/O	74
5.	Timing and Synchronization	74
C.	RPS PROFILER	74
D.	INPUT ERRORS	75
V.	TESTING AND RESULTS	77
A.	METHODOLOGY	77
1.	Benchmarks	77
2.	Multiprocessor Topology	78
3.	Scheduling Methods	78
B.	EXPERIMENTS	78
1.	Accuracy of the System Model	78
2.	Efficiency of the Scheduling Heuristic	79
a.	Random Task Graphs	80
b.	Correlator Graph	84
c.	Gaussian Elimination	87
3.	Comparison of the Two Scheduling Variations	90
4.	Comparison with MH heuristic	91
a.	Random Task Graphs	91

b.	Correlator Graph	93
c.	Gaussian Elimination	94
VI.	CONCLUSIONS	97
A.	CONCLUSIONS	97
B.	FUTURE WORK	98
APPENDIX	99
A.	RPS SCHEDULER AND RPS PACKAGER MAIN PROGRAM	99
B.	SCHEDULE FUNCTIONS HEADER FILE	99
C.	SCHEDULE FUNCTIONS SOURCE FILE	101
D.	PACKAGING HEADER FILE	119
E.	PACKAGING SOURCE FILE	120
F.	RPS PROFILER MAIN PROGRAM	138
G.	PROFILER HEADER FILE	138
H.	PROFILER SOURCE FILE	139
LIST OF REFERENCES	143
INITIAL DISTRIBUTION LIST	145

I. INTRODUCTION

The physical limitations of traditional processor architectures cause an upper bound on the attainable performance of the processor. A method of increasing the performance of these processors is to build a multiprocessor system such that different segments of a program can be run in parallel.

In order to utilize this type of system, the program must be broken up into relatively independent parts and each part assigned to a processor. The problem of deciding which processor to assign which piece of the program is of prime importance. The processor assignment must be made in such a manner as to achieve a desired level of performance. This is known as the mapping problem. [HAM 92]

Since it has been proven that finding an optimal solution to the mapping problem is NP-complete, research into solving this problem has led to the development of many suboptimal solutions. Suboptimal solutions take on many forms including graph-theoretic, mathematical programming, queuing theory, search algorithms, and heuristics. These methods, while not necessarily providing the best processor assignment, provide a solution which results in a satisfactory assignment. [KAS 94]

A. OBJECTIVES

The work presented in this thesis deals with the mapping of iterative applications. An area which provides many applications of this type is Digital Signal Processing (DSP). In DSP applications, data arrives periodically requiring the program to be executed on each instance of data.

An application that this thesis would be concerned with are represented by acyclic data flow (or task) graphs. This graph is a directed acyclic graph (DAG) in which the nodes of the graph represent the tasks of a program and the edges represent the communications between the tasks. By requiring that the graph be acyclic, the condition that the processing of new data does not depend on earlier results is established. An example of this type of graph is shown in Figure 1.

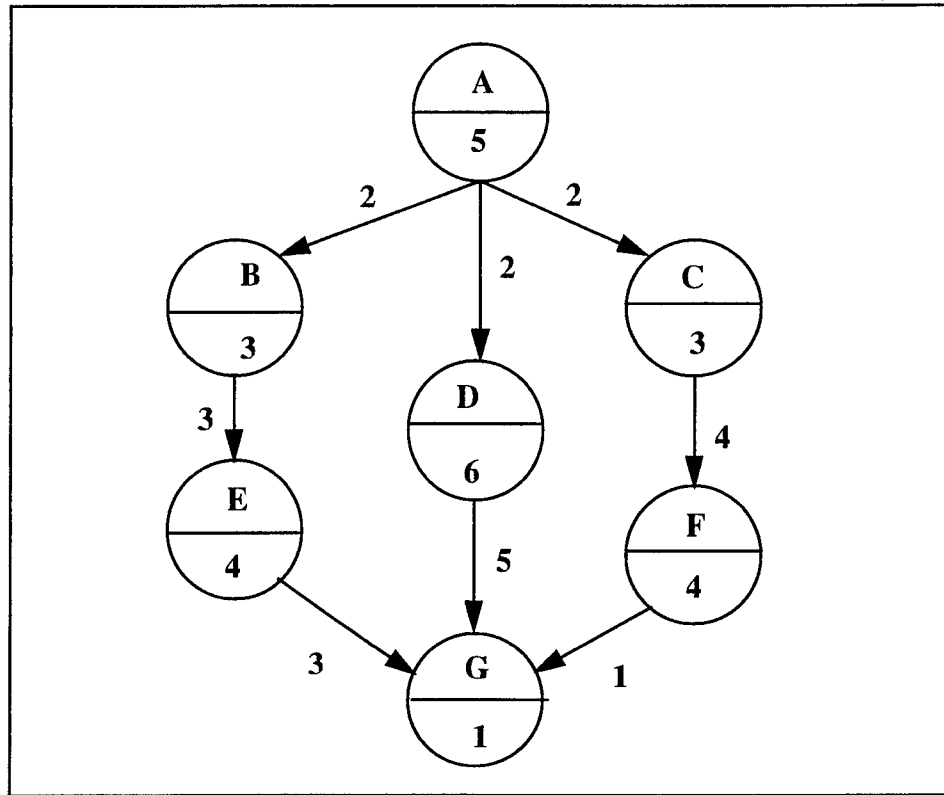


Figure 1: Sample Task Graph. From [KAS 94].

Some task graphs that exhibit this characteristic are able to take advantage of pipelining provided the processing of new data does not depend on earlier results. Task graphs that are pipelined may be executed by overlapping successive instances such that execution begins on later arriving data before execution is completed on earlier data. [HOA 93]

Prior work in this area has been concerned with minimizing the execution time of one instance of the program. [BOK1 81, BOK2 81, DIX1 93, DIX2 93, ELR 90, HOA 93, KER 70, LOV 88]

The Periodic Scheduling (PS) heuristic developed by Kasinger [KAS 94] is designed to maximize the throughput of applications of this nature. It maps the tasks of a task graph to the processors of a system of any topology. These tasks are represented as nodes of a

directed acyclic graph. It considers communication between tasks, resource contention, and system topology in making processor assignments.

The PS heuristic gives a general heuristic for assigning repetitive task graphs to processors on a distributed memory multiprocessor system. Since PS is not designed for any particular multiprocessor, the heuristic assumed a simplistic computation-communication model for the underlying parallel processing system. In order to determine the validity of PS as an effective method of mapping repetitive task graphs to multiprocessors, the actual computation-communication model of the system utilized must be incorporated in the heuristic.

1. Scope of the Thesis

This thesis presents a Realistic Periodic Scheduling (RPS) heuristic for an INMOS T800 Transputer system utilizing Parasoft Express software. The RPS heuristic is a modified version of the PS heuristic which accounts for system characteristics which have been ignored in PS. The characteristics of the parallel processing system used had to be determined experimentally by employing several tests designed to explore various aspects of the systems communications model.¹ Tools were developed to aid the programmer in determining the schedule, packaging the nodes into compilable source code, and determining accurate values for a nodes computation time. Experiments are conducted on various benchmark programs using different system topologies and various communications/computation ratios. The performance is compared with predicted performance.

B. THESIS ORGANIZATION

Chapter II describes the mapping problem, highlights relevant prior research and describes the PS heuristic and its characteristics. Chapter III presents a description of the hardware and software systems utilized. In Chapter IV, the RPS heuristic is presented along

1. System characteristics were determined experimentally due to a lack of available information describing the computation-communication model of the system.

with the programming tools developed. These tools are the RPS scheduler, RPS packager, and RPS profiler. Modifications of the PS heuristic to reflect the system model are also discussed. Research methodology and experimental results are explained in Chapter V. Chapter VI draws conclusions from the results and suggestions for future work are given.

II. BACKGROUND

A. THE MAPPING PROBLEM

A distributed memory multiprocessor consists of multiple processors, each having its own memory, connected by communications links. This system may be represented by an undirected graph in which the processors are the nodes and the communications links are the edges. We assume that the processors are homogeneous¹ such that task execution time does not depend on the processor assigned. The communications links are assumed to be homogeneous and bi-directional. An example of a processor graph is given in Figure 2.

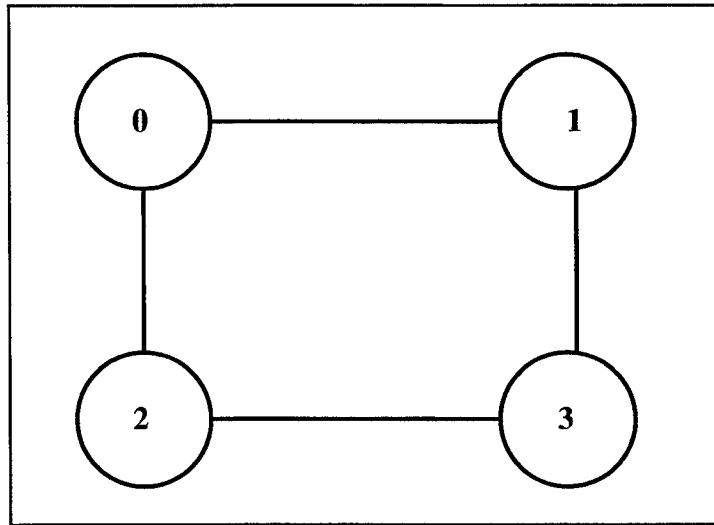


Figure 2: Sample processor graph. From [KAS 94].

In our model, a parallel program is a set of tasks which communicate data between them. Each task consists of three phases. These are consume inputs, execute, and produce outputs, and are carried out in this order. These tasks can be represented by a weighted digraph. The weights represent the amount of computation and communication that is associated with each task. The task graph consists of the tasks (nodes) and the communications annotation (edges). The applications that are of concern to us have the

1. This constraint simplifies the exposition, but can be lifted without serious effects on our methodology.

additional property that the graph must be acyclic. An example of a task graph is given in Figure 3. [HAM 92]

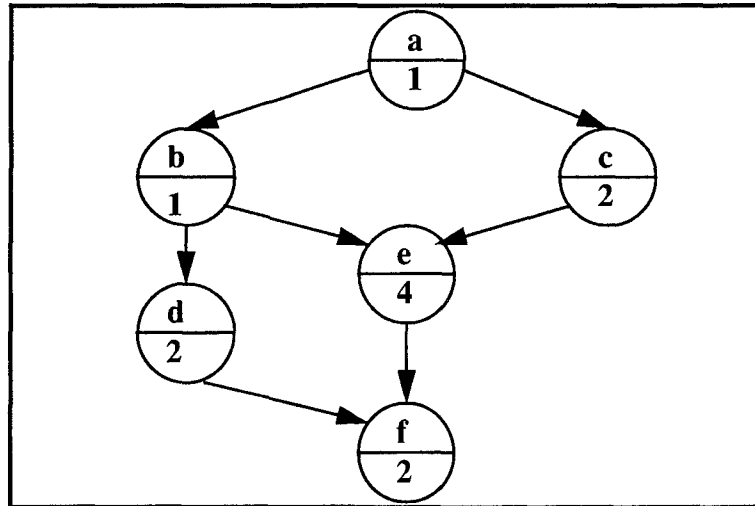


Figure 3: Sample task graph. From [AKI 93].

The mapping problem consists of finding a mapping of tasks from the task graph to the processors in the system such that we minimize the execution time of the program. In order for a mapping algorithm to be practical, less time should be devoted to the mapping problem than to executing the application. Since it has been proven that finding an algorithm which solves the mapping problem is NP-complete for most interesting instances, methods which give sub-optimal solutions have been developed. [HOA 93]

B. PRIOR WORK

Heuristic methods for solving the mapping problem can lead to an acceptable solution in a reasonable amount of time. A heuristic contains three parts: (1) an initial guess at the solution; (2) an improvement procedure; (3) an objective function. These can be broken down into one-pass and iterative. A one-pass heuristic makes a processor assignment and does not change it. An iterative heuristic temporarily reassign tasks to different processor and examine if the solution has improved. This continues until an acceptable mapping is found or the maximum number of iterations has taken place. [HAM 92]

Iterative heuristics are either deterministic or probabilistic. Deterministic-iterative algorithms evaluate an objective function after each iteration and the solution is kept only if it is better than the previous one. Probabilistic algorithms keep the solution if it is better than the previous. Solutions that are worse than previous are also kept according to some probability. [HAM 93]

1. Graph Partitioning

Graph partitioning algorithms relate to graph mapping. A graph partitioning breaks a graph into parts. Each part can then be embedded into the processors graph. This can be viewed as a one-pass heuristic. [HAM 93]

One such graph partitioning algorithm is presented by Kernighan and Lin [KER 70]. There algorithm deals with partitioning a graph into two equal sized subsets of the nodes. It begins with an arbitrary partition of the graph, and then, by exchanging nodes of the subsets, attempt to reduce the cost of the edges cut. The complexity of this algorithm is $O(n^2)$. This algorithm is also extended to include partitioning into unequal sized subsets and multiple subsets. Multiple subsets are found using repeated applications of the two subset partition.

2. Mapping

Bokhari [BOK1 81] presents a heuristic for mapping task graphs onto a Finite Element Machine (FEM). This method takes the adjacency matrix of the task graph and, through a series of pairwise exchanges, outputs a permutation of this matrix that more closely resembles the adjacency matrix of the FEM. These exchanges are made such that the exchange that results in the largest gain in cardinality. The cardinality is defined as the number of task graph edges that fall on array edges. If no such exchange exists, a probabilistic jump to a mapping close to the current is made in an attempt to improve the solution.

Bokhari [BOK2 81] also presents a dynamic programming approach to solving the mapping problem. He assumes that the processors in the system are dissimilar. Because of

this, the tasks will take different amounts of time to execute on different processors. The objective of the algorithm is to assign the tasks to the processor on which they execute most rapidly whenever possibly, while also considering overhead due to interprocessor communication. His algorithm uses a shortest tree approach and minimizes the sum of node execution time and interprocess communications cost. One limitation to this method is the communications pattern of the task graph forms a tree. This algorithm finds a solution in $O(mn^2)$ time where m is the number of tasks and n is the number of processors.

Lo [LOV 88] developed a three part algorithm which minimizes the total execution and communication costs of the processor assignment. It seeks greater concurrency and load balancing of the tasks by including the effects of interference caused by tasks being assigned to the same processor. Interference costs include contention for resources, communications costs associated with contention for buffers and synchronization, and costs due to tasks being assigned to the same processor. This algorithm, which uses static assignment of tasks to processors, consists of a grab phase, a lump phase, and a greedy phase. The grab phase views the n -processor system as a two processor system, a given processor as one and all of the rest as the other. A Max Flow/ Min Cut algorithm is used to assign tasks to the single processor. The tasks that do not get assigned by this method are sent to the lump phase where all remaining tasks are assigned to one processor if it does not result in too high of a cost. If tasks still remain, the greedy phase clusters tasks with high interprocess communications and assigns them to processors to the cheapest processor for that cluster.

A heuristic that maximizes throughput is presented by Hoang and Rabaey [HOA 93]. This heuristic attempts to maximize the parallelization and pipelining of a program. They schedule nodes to processors such that each processor belongs to a stage of the computation. The algorithm attempt to minimize the length of a stage. By allowing more than one processor to be assigned to a stage, the parallelization of the program can be exploited. Nodes are scheduled to start as soon as possible taking into account communications delays, memory capacity and processor availability.

The next methods presented are closer to what we are trying to achieve. These algorithms utilize the task graph in order to determine the schedule. These differ in that they are designed to minimize execution time.

Dixit-Radiya and Panda [DIX1 93] utilize a Temporal Communication Graph (TCG) to represent the task graph. In this graph, a task is defined to be a group of nodes of the graph. They attempt to find a minimal completion time of the program by minimizing link contention. Temporal link contention and unequal distances between processors are both considered. Their heuristic generates an initial processor assignment by selecting tasks in decreasing order of total communications, minimizing the distance between heavily communicating tasks. Once this is done, the iterative step does a pairwise exchange of tasks which reduce the maximum link contention.

Dixit-Radiya and Panda [DIX2 93] present three other heuristics using the TCG. These deal with clustering of tasks. With these heuristics, the tasks are already assigned to processors. The desire is to cluster tasks together on processors such that a reduction in completion time is achieved. One chooses tasks to merge by examining each pair of tasks and picking the pair that results in the greatest reduction in completion time. The other two also include processor contention in the heuristic by examining the amount of parallelization between clusters and making a trade off between this the communication between clusters.

The Mapping Heuristic (MH) was devised by El-Rewini and Lewis [ELR 90]. This method attempts to minimize final completion time by taking into consideration the target machine, communications delays, contention, and the balance between computation and communications. List scheduling is used as each node of the task graph is assigned a priority. The tasks are placed on an event list if they have no predecessors. The first ready task is removed from the list and scheduled such that it cannot finish any earlier on another processor. If the task has immediate successors, the status of the successors is updated. This continues until all tasks have been scheduled.

The next technique is extremely close to our method. More time will be devoted to explaining it since it strongly relates to our algorithm.

The Revolving Cylinder technique by Shukla, Little, and Zaky [SHU 92] is a method of determining task execution sequence in repetitive applications. It was originally designed for shared memory multiprocessors. This method takes a task graph as its input and produces a mapping to the systems processors as an output. The technique gets its name from the form that the schedule takes on. If the resulting schedule were to be wrapped around, with the end touching the beginning, it would form a cylinder. The cylinders circumference is determined by summing the total execution time of all nodes and dividing by the number of processors. This is representative of the maximum throughput that the system will support.

Each processor in the system is assigned one band of the cylinder. Each of these bands is then divided into slots. The nodes of the task graph are then assigned to the different bands by fitting them into the different slots. These assignments can be made according to any method desired. [SHU 94]

The resulting cylinder is the schedule for one instance of the task graph. In the case of a periodic application, when data is periodically arriving, the task graph is instantiated each time new data arrives. Subsequent instances of the graph are overlapped with previous instances. Each node is assigned an index to prevent conflicts between separate instances of the graph. This is necessary since dependencies in the associated task graph require different instances of the task graph to be mapped to one revolution of the cylinder. [SHU 94]

The result of RC scheduling is shown in Figure 4. This figure shows the RC scheduling of the graph in Figure 3 on a two processor system.

Using Figure 4 as an example, we see that task **a** is executed first on processor 1. At time 1, task **f** is scheduled to execute on processor 1, but since it is dependent on tasks that have not yet completed, it cannot execute. Similarly, task **e** was scheduled to execute at time 0 on processor 2, but could not for the same reason. At time 3, task **c** executes on

processor 1 followed by task **b** at time 5. These can execute since they are dependent on task **a** which completed execution at time 1. The second iteration of the cylinder begins at time 6. Task **a** executes again as shown by the increase in its index from 0 to 1. Since task **e** is dependent on tasks **b** and **c**, it can now begin execution. Since it is for iteration 0 of the graph, it has an index of 0. At time 7, task **f** still cannot begin since it is dependent on **e** which has not finished. The figure shows that **f** executes during the next iteration of the graph. This illustrates how each iteration of the cylinder contains tasks belonging to different iterations of the graph. It also shows how the cylinder works. As time progresses, the cylinder turns. If the task that is on the cylinder at that time can begin, it is executed. If not, That task waits until the next time the cylinder reaches that point an executes then if it can.

3. Summary

Many methods of mapping tasks to processors have been developed. Most of these attempt to minimize the completion time of the application as opposed to maximizing the throughput. The one that does have the goal of maximizing throughput does not take into account system topology or resource contention.

C. PS HEURISTIC

The goal of the PS heuristic developed by Kasinger [KAS 94] is to maximize the throughput of an executing process since the processes which are of interest are repetitive in nature. This way, the maximum number of iterations of the process may be completed in the minimum amount of time. While this may lead to a longer completion time for any one particular instance, this is only a concern if there are real time constraints for completion time.

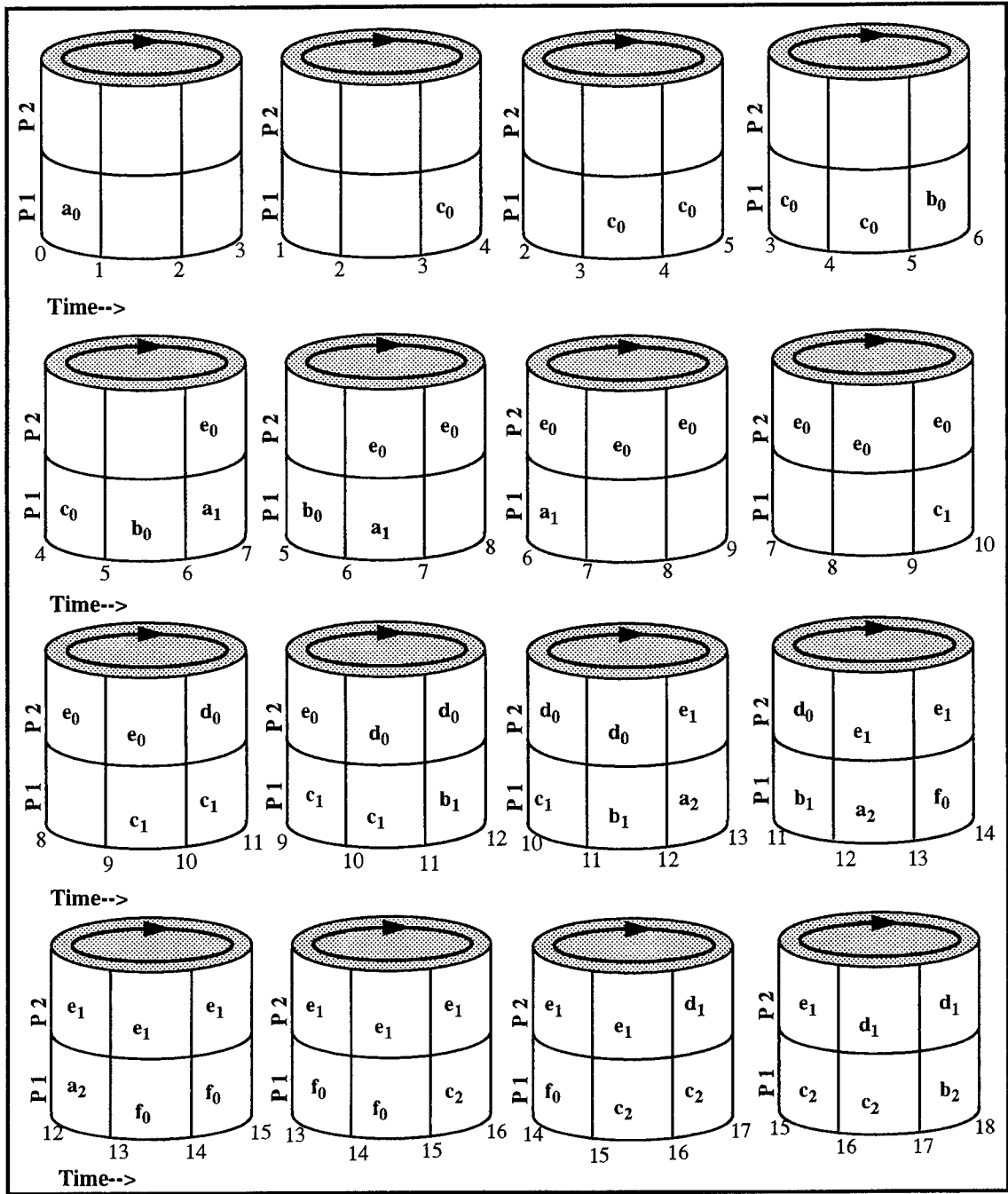


Figure 4: Revolving cylinder schedule applied to task graph in Figure 3.
From [AKI 93].

Kasinger's PS heuristic uses the characteristics of revolving cylinder to schedule successive instances of the task graph. PS differs from revolving cylinder in that it is designed for a distributed memory multiprocessor and a specific heuristic is used for assigning tasks to processors. This heuristic considers communications, contention, and the interconnection network between processors in making its assignment of nodes to processors. The PS heuristic can be used for scheduling task graphs on systems with any system topology and any processor speed.

1. Processor Selection

The goal of the PS heuristic is to maximize the throughput of a repetitive task graph. This goal is accomplished by scheduling the nodes of the task graph to the processors of the system in such a way that the maximum usage of the system's resources is minimized. These resources consist of the processors and the communication links connecting them. A Processor Utilization Table is used to hold information on the level of utilization of the processors and a Link Utilization Table contains information on the utilization level of the communication links. [KAS 94]

The Processor Utilization Table is simply an array that consists of one array element corresponding to each processor in the system. The Link Utilization Table is a two dimensional matrix. Each row and column of the matrix corresponds to a link between two processors in the system. If no link exists between two particular processors, the corresponding matrix location is unused. [KAS 94]

The PS heuristic works in the following way. When a task is ready to run, the task and copies of both utilization tables are passed to the PS procedure. Copies of the utilization table are used in testing different processor assignments for the task. The task is tested on each processor in the system. The Processor Utilization Table is updated to reflect how the assignment of the task to the given processor affects the utilization level of that processor. [KAS 94]

Since the ready task may have predecessors that are assigned to a different processor, the Link Utilization Table must be updated to reflect the communication that will be necessary using this processor assignment. After all communication requirements are added to the Link Utilization Table, the tables are search to find the maximum utilization of any resource. This information is stored and used to compare against other processor assignments for the task. The first “best” mapping is the one selected. This means that the task will be assigned to the lowest numbered processor of the ones with lowest maximum resource utilization. [KAS 94]

This procedure is repeated for each task in the task graph. The Processor Utilization Table and Link Utilization Table are updated to reflect the assignment of all tasks that have been previously assigned to processors. As a new task in the graph is ready to run, PS is executed with the new utilization tables. [KAS 94]

2. Instance Overlap

Kasinger’s PS heuristic uses the revolving cylinder technique to maximize the throughput of repetitive applications. The use of the revolving cylinder technique allows for the possibility of overlap in the execution of instances of the task graph. By overlapping instances of the task graph, different processors may be working on different instances of the graph at any given time. This feature allows us to more efficiently utilize the processor of the system.

After PS has determined the processor assignment for the tasks of the graph, the task is assigned to the processor at the earliest time that it is available. This has the effect of compacting the execution closer to the start of the cylinder which leads to a shorter execution time for one revolution. Since precedence relationships may exist between tasks, compacting the tasks may cause tasks that rely on data from other tasks to be scheduled before the data is available. When this occurs, each cylinder revolution contains tasks that belong to different instances of the task graph. [KAS 94]

In order to account for different instances of the task graph executing during each iteration of the cylinder, indices must be assigned to the nodes. These indices represent which instance of the task graph the nodes belong to. This comes from an analysis of the precedence relations between the tasks. [KAS 94]

3. Example

An example is provided to illustrate how PS works. The task graph in Figure 5 has been scheduled using PS on a four processor ring and the resulting schedule is shown in the Gantt chart in Figure 6. The resulting processor utilization and link utilization are given in Figure 7.

As can be seen from the Gantt chart, the throughput of this task graph is 6 time units. This means that every 6 time units, a new iteration of the task graph can be initiated, and a previous iteration completes execution. By examining the indices of the tasks, the latency of one iteration of the task graph can be determined. Task A is the starting task in the graph and it has an index of $i-2$. Task G is the last task to be executed and has an index of $i+1$. This means that any instance of the task graph will complete execution after four cylinder iterations from when it began. Since the length of the cylinder is 6 time units and task G finishes at 3 time units, the latency of this task graph is 21.

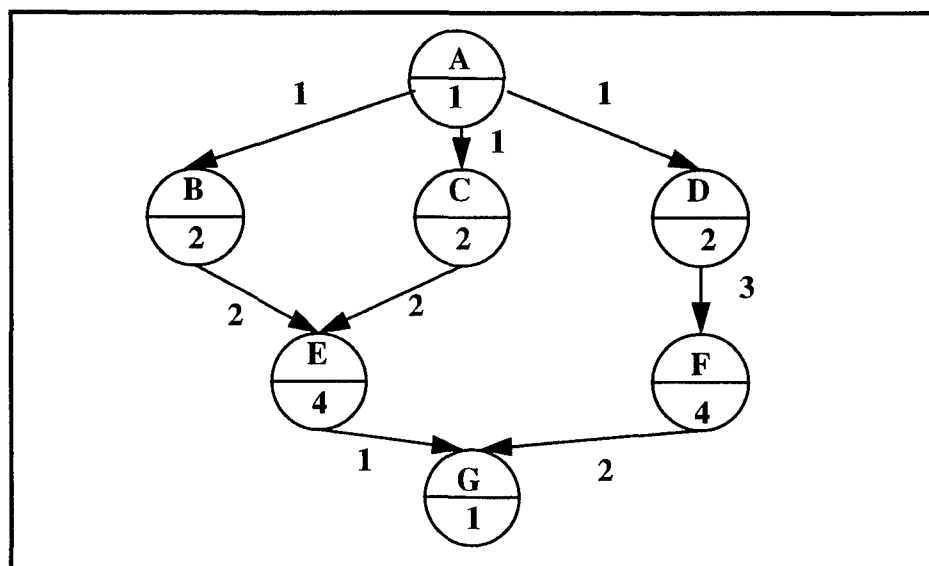


Figure 5: Sample task graph. From [KAS 94].

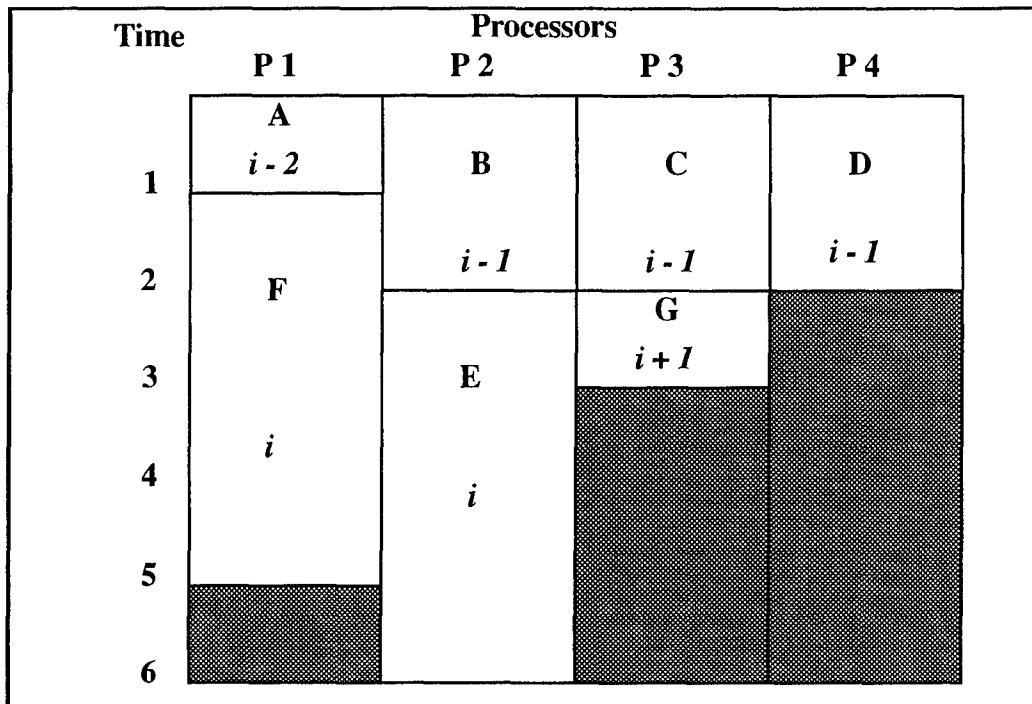


Figure 6: Gantt chart for schedule developed with PS. From [KAS 94].

P1	P2	P3	P4
5	6	3	2

Processor Utilization Table

	P1	P2	P3	P4
P1	---	3	3	---
P2	2	---	---	1
P3	1	---	---	0
P4	---	1	0	---

Link Utilization Table

Figure 7: Processor Utilization and Link Utilization tables produced by PS for scheduling task graph in Figure 5. From [KAS 94].

III. HARDWARE AND SOFTWARE CHARACTERISTICS

A. INMOS TRANSPUTERS

“Transputer” comes from the words *Transistor Computer*. It is a microcomputer with its own local memory and link that enable one Transputer to be connected to others.

A typical Transputer is a single chip containing a processor, memory, and communication links that provide point-to-point connection between Transputers. In addition, each Transputer contains special circuitry that allow it to be adapted to a particular use. An example of this would be a peripheral control Transputer such as a graphics or disk controller. [INM 89]

Transputers can be connected in a network or used in a single processor system. Through the use of the communications links, it is an easy task to connect a group of Transputers into a network. Using the Transputer in a single processor system is just as simple as ignoring some of the communications links.

1. General Information

The Inmos Transputers that are used in this research are the IMS T800 Transputers. These processors consist of the CPU, four link interfaces, on-chip RAM, and a memory interface. The link interface supports a standard link communications frequency of 10 Mbits/sec. Each Transputer supports concurrent execution through a microcoded scheduler. [INM 89]

Communications in the Transputer are point-to-point, unbuffered, and synchronized. This means that all communications are “blocking”, both the sender and receiver must be ready to communicate or the processes waits. The communications are handled by means of channels. A channel between processes on the same Transputer is implemented as a single word in memory. Channels between Transputers are established on the point-to-point links. [INM 89]

2. IMS T800

The T800 Transputer is a 32 bit microprocessor. It has 4 kbytes of on-chip static RAM memory and supports 4 Gbytes of directly addressable external memory. Its communications links operate on the standard 10 Mbits/sec frequency, and also support 5 and 20 Mbits/sec communications. A 64 bit floating point unit is also located on chip for floating point calculations. A block diagram of the T800 is shown in Figure 8. [INM 89]

Processes running on the T800 can run at either priority 1 (low), or priority 0 (high). High priority processes are expected to run for only a short period of time. If multiple high priority processes are ready, one is selected and runs until it is waiting for communications, a timer input, or completes. If no high priority processes are able to proceed, a low priority process is selected. Low priority processes are time-shared to provide an even distribution of processor time. [INM 89]

3. IMS B003

The Inmos B003 Transputer board is a board that consists of four Inmos Transputers. These Transputers can be of any type desired. They are hard mounted on the board in a four processor ring configuration [INM 86].

The ring configuration on the board is established by hard-wiring a communications port from one Transputer to a communications port of another. These hard-wired links cannot be disconnected. The connection scheme that is used to connect the Transputers is to establish a link between the number two communications port of each Transputer and the number three port of the respective successor Transputer. In other words, Transputer zero's number two port is connected to Transputer one's number three port and so on. Transputer zero is considered to follow Transputer three in the ordering. This configuration is shown in Figure 9. [INM 86]

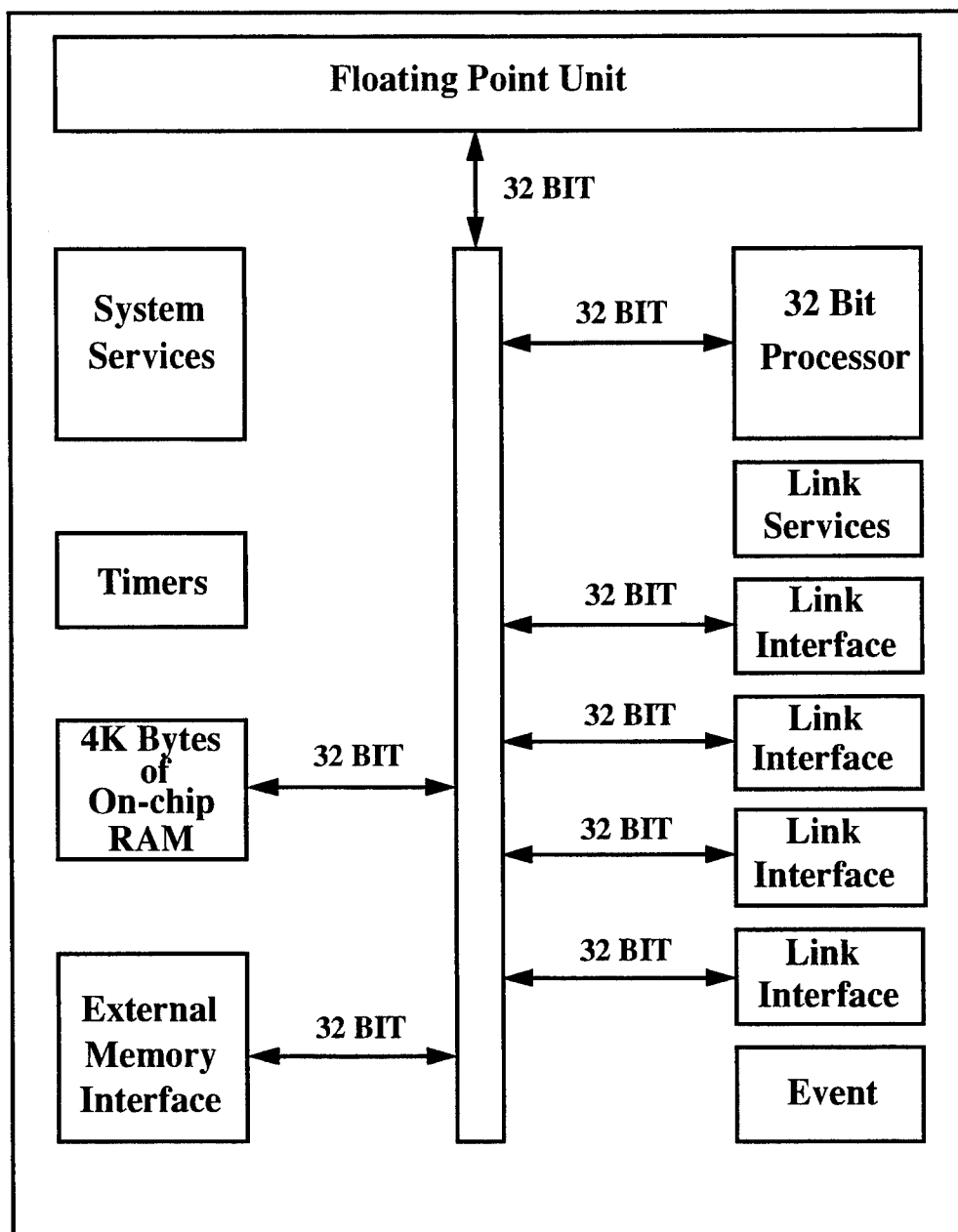


Figure 8: Block diagram of the IMS T800 Transputer. From [INM 89].

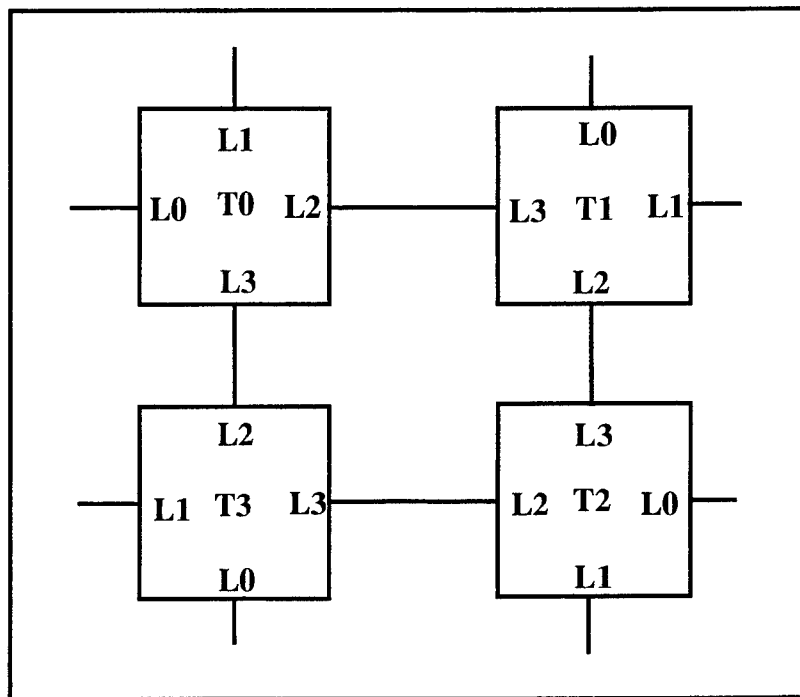


Figure 9: IMS B003 ring configuration. From [INM 86].

Connections using the other communication ports, either between processors on the same board or processors on other boards, is made by using wire jumper links to manually connect the ports [INM 86]. These connections can be used to create any processor configuration using any number of Transputers. Figure 10 shows a block diagram of the link connection pins for the unconnected links.

When more than one board is used, reset signals for the board must also be sent to all boards being used. This is done by connecting the “down” port of each board to the “up” port of each successive board. Any number of boards can be daisy chained together in this manner. [INM 86]

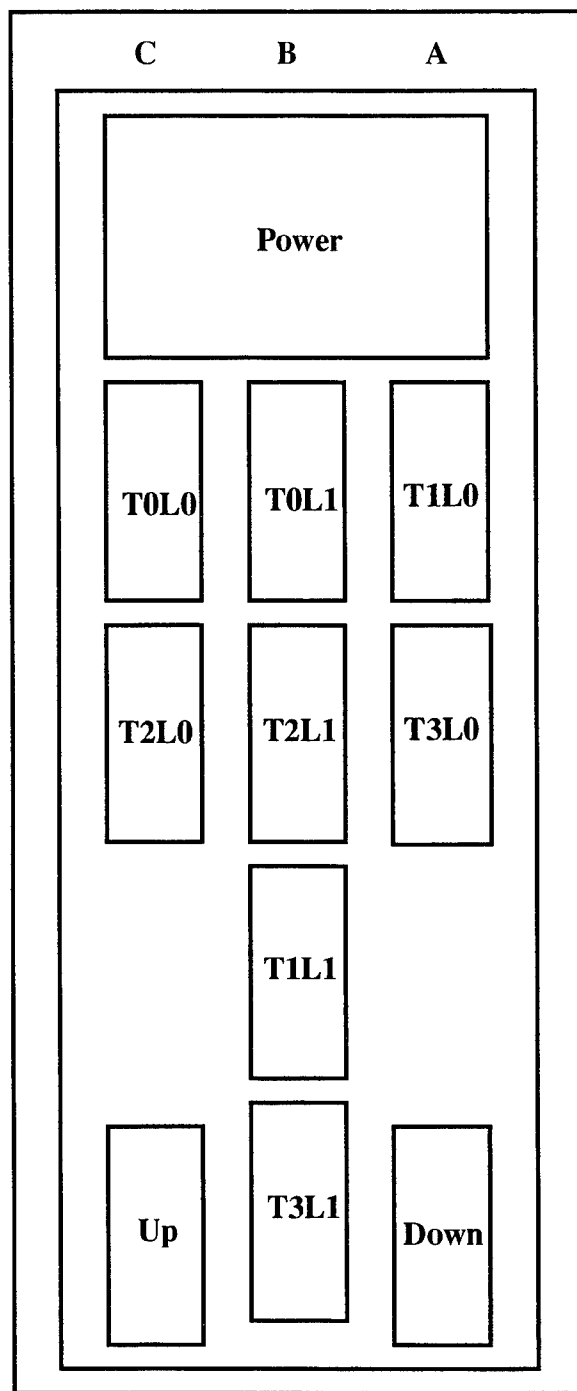


Figure 10: IMS B003 pin connectors. From [INM 86].

4. IMS B004

The Inmos B004 Transputer board is a board that is mounted internally in a DOS based computer. This board is used to give the DOS computer access to the Transputer system so that it can function as the host machine for the Transputer. The board has one T414 Transputer hard mounted [INM 85]. A block diagram of the B004 Transputer board is shown in Figure 11.

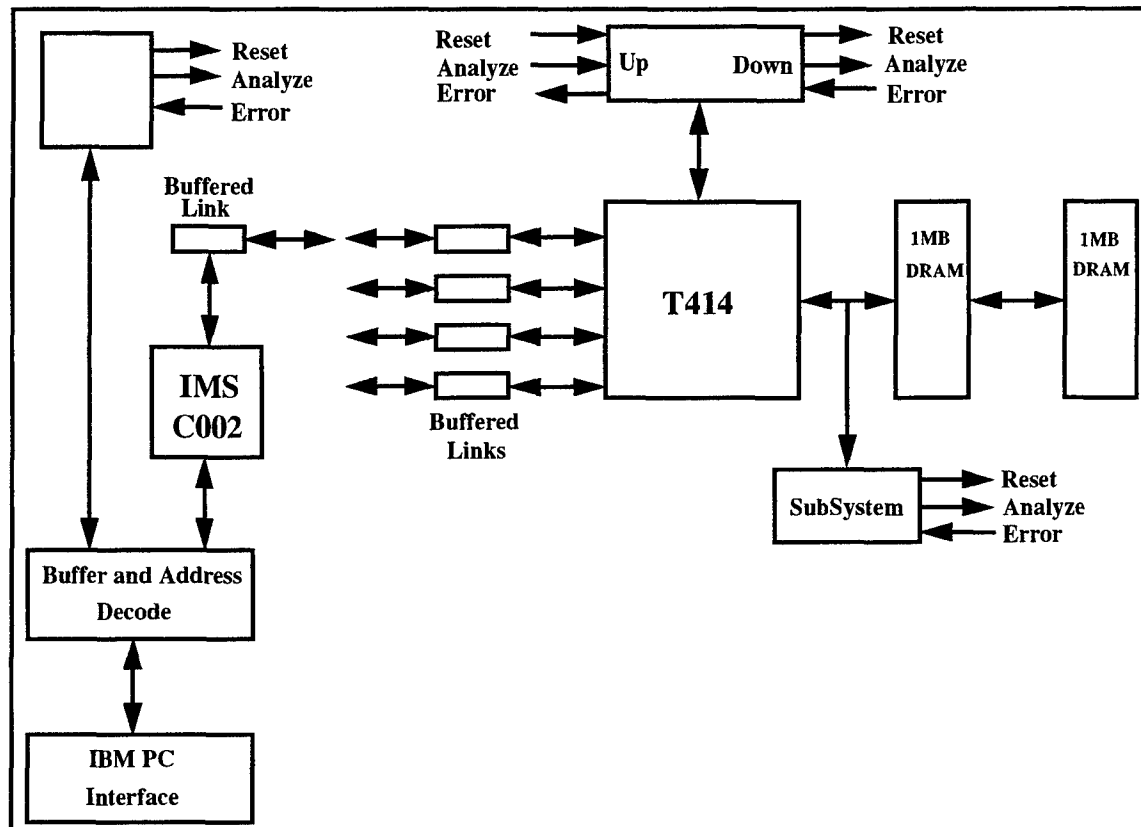


Figure 11: Block diagram of the IMS B004 board. From [INM 85].

The DOS computer is connected to the Transputer system by using a wire link jumper to connect the PC link port on the B004 board to the zero link of the boards T414 Transputer. Reset signals are sent from the PC to the Transputer system by connecting the PC system port to the up port on the B004 board. This is done using the wire reset jumper. Figure 12 shows a block diagram of the connector pins for the board. [INM 85]

Subsequent boards in the system are added by connecting the number one, two, and three links of the boards T414 Transputer to links of other Transputers in the system. Reset signals are sent to the other Transputer boards by connecting the B004's down port or subsystem port to the up of the up port of the next board. The boards are then daisy chained together. The difference between the down port and the subsystem port is that the down port will cause all boards to be controlled by the PC where the subsystem port will allow other boards to control its' own system of boards. Daisy chaining of Transputer boards is shown in Figure 13. [INM 85]

The T414 Transputer can be bypassed on the B004 board through the use of two ground wires. Pin b6, the NotLink pin, is connected to pin a13, the GND pin. Pin b27, the NotSystem pin, is connected to pin a17, the GND pin. This enables the T414 processor on the B004 board to be bypassed so that the PC can be directly connected to a Transputer in the system. This enables the system to consist entirely of one type of Transputer since the B004 board is always fitted with a T414 Transputer. Figure 14 provides a description of the B004 board connection pins and indicates the proper connections to achieve this. [DUN 94]

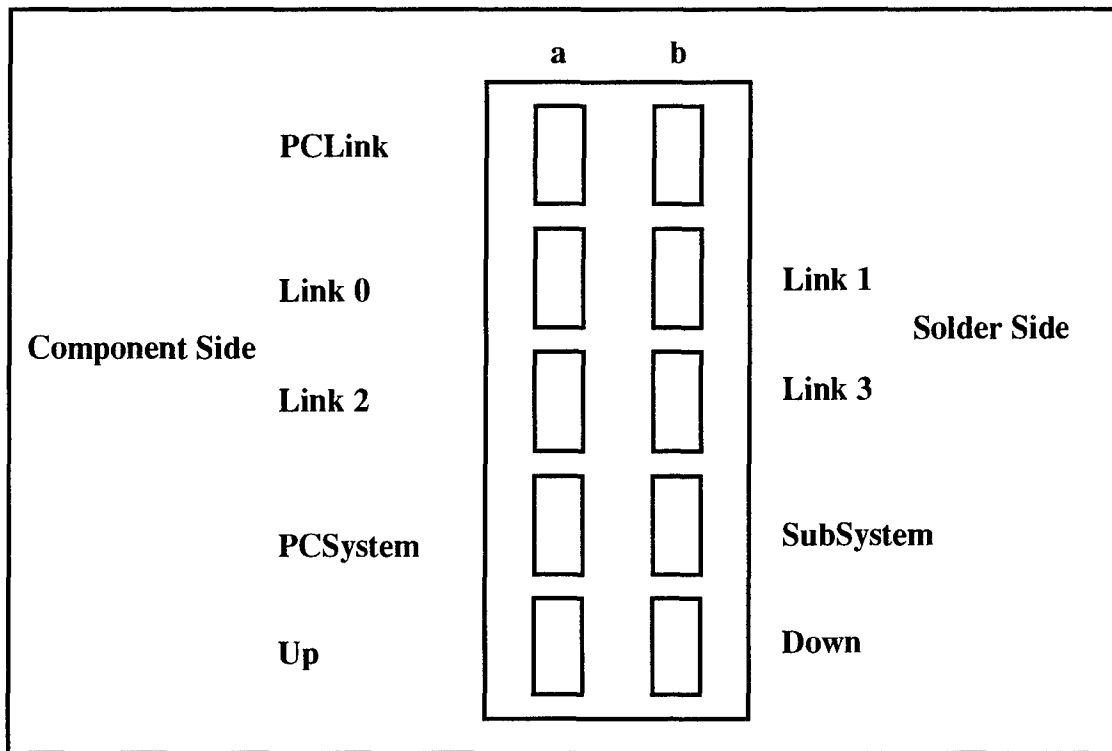


Figure 12: IMS B004 pin connectors. From [INM 85].

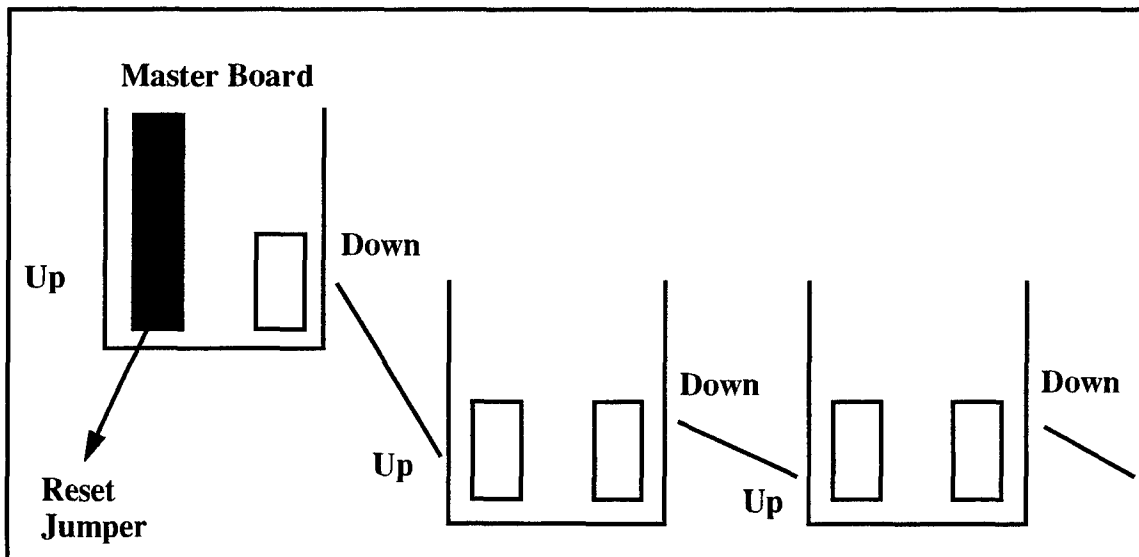


Figure 13: Daisy chaining of subsequent boards. From [INM 85].

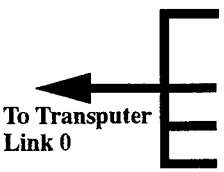

	Pin	b	a
	1	GND	NC
	2	(missing)	(missing)
	3	PCLinkOut	NC
	4	PCLinkIn	NC
	5	GND	NC
	6	NotLink	NC
	7	GND	GND
	8	(missing)	(missing)
	9	LinkOut 0	LinkOut 1
	10	LinkIn 0	LinkIn 1
	11	GND	GND
	12	(gap)	(gap)
	13	GND	GND
	14	(missing)	(missing)
	15	LinkOut 2	LinkOut 3
	16	LinkIn 2	LinkIn 3
	17	GND	GND
	18	(gap)	(gap)
	19	(gap)	(gap)
	20	(gap)	(gap)
	21	(gap)	(gap)
	22	PCNotReset	SubsystemNotReset
	23	PCNotAnalyse	SubsystemNotAnalyse
	24	PCNotError	SubsystemNotError
	25	GND	GND (missing)
	26	(missing)	(missing)
	27	NotSystem	NC
	28	UpNotReset	DownNotReset
	29	UpNotAnalyse	DownNotAnalyse
	30	UpNotError	DownNotError
	31	GND	GND (missing)
	32	GND (missing)	GND (missing)

Figure 14: B004 board pin connectors with connections for bypassing mounted T414 Transputer. From [DUN 94].

B. PARASOFT EXPRESS

The Parasoft Express software is a parallel programming tool that provides message passing features which enable the user to create parallel applications. It cooperates with the existing operating system to provide a familiar environment to the user. It is through this environment that Express allows the user to access the parallel processing system. [PAR 90]

Express is itself not a programming language for parallel systems. Rather, it provides compilers for various existing programming languages that utilize libraries that provide access to the parallel features of the system. [PAR 90]

The Express kernel provides the basic functionality that is needed by parallel programs. Once the kernel is loaded, the system can utilize Express communications primitives to allow the processors to work together. These primitives include both high and low level message passing. The kernel also provides various parallel programming tools such as a performance analyzer, a debugger, multitasking features, and parallel graphics.

1. System Configuration

In order for the Express kernel to operate properly, it must know the configuration of the system. This is done through the use of a utility called "cnftool". This utility creates files that are used by the kernel to establish routing and send reset signals. The files contain a description of the interconnection between the processors in the system, information about the way in which reset lines are connected, and message forwarding information. [PAR 90]

Express uses a specific terminology to describe the parallel processing system. This terminology will be used throughout the description of Express. The PC that the Transputer system is connected to is referred to as the host. It runs a program known as the host program. Each processor in the parallel system is referred to as a node and the programs that run on these processors are node programs.

When "cnftool" is run, a "worm" can be run which will detect all mechanical links that are present in the current system. This can be used of the configuration can be plotted

manually, If a link that is specified in the information files is not present in the system, the system will fail to work properly. The worm can be used to avoid this problem. [PAR 90]

The worm cannot detect links that are provided by a C004 switch. These links must be manually entered using the "cnftool". [PAR 90]

Once the nodes and links of the system are configured, a forwarding table and reset tree must be created. The forwarding table creates a routing scheme that will be user by the system. Express offers three choices, hypercube, torus, and general. The hypercube and torus options will ensure routing such that deadlock will not occur. The general routing can be used for all configurations that are neither a hypercube nor a torus. This will provide routing using shortest path and is not guaranteed to be deadlock free. The routing of the system is static. In other words, once the kernel has been loaded to the system, the routing path between any two nodes is fixed according to the established forwarding table. [PAR 90]

2. Communications Functions

The Express programming model used for this thesis is the host-node model. The host PC runs a program that communicates with the programs running on the nodes of the parallel system. This results in the host program having access to the features of the host machine, while the node programs have access to only those features provided by the parallel system. This means that all I/O must be performed by the host program with data being sent to or received from the node programs. This model also provides the ability of different node programs being executed on the various processors of the system simultaneously.

In order to utilize the host-node programming model, the user must write a host program that performs the functions which will control the node programs. These function include processor allocation, program loading, and communications with the nodes for I/O functions.

a. Processor Allocation

Processor allocation is achieved using the “exopen” function. This function allocates the number of processors requested, if available, to the requesting host. The function assigns an index to a group of processors which distinguishes them as being allocated to a particular host. These processors can then be loaded with node programs by the host. [PAR 90]

b. Loading Programs

The node programs are loaded by using the “exload” or “explode” functions. These functions load the node programs onto the designated processors. The “exload” function loads all processors in the group with the designated node program. The “explode” function is used to load different node programs on the processors in the group. “Explode” loads a given processor with a designated node program. This function can be called multiple times enabling each processor to be loaded with any node program desired. While “Explode” can be called multiple times, multiple calls to “explode” will result in the second node program to be written over the first. [PAR 90]

Once the node programs are loaded, they must be given a command to begin execution. If the “exload” function is used, all node programs will begin execution after they are loaded. If the “explode” function is used, a separate function must be used to start execution of the node programs. This is done through two functions, “exstart” and “exmain”. “Exstart” is used to load additional arguments to the node programs. It can be used to load arguments to any or all loaded node programs. Once the arguments are loaded, a call to “exmain” tells the node programs to begin execution. This can be used to start execution of any or all node programs. [PAR 90]

c. Message Passing

While the processors are executing their node programs, the programs will need to communicate with each other and the host. The Express kernel is based on an asynchronous, point-to-point message passing system. This allows a message to be sent

from any node to any other node and the kernel is responsible for buffering and routing of messages. Since the Transputers themselves do not support asynchronous message passing, the Express kernel provides it. Asynchronous message passing is provided through the functions “exwrite” and “exread”. [PAR 90]

These two functions are the basic communications functions for a packet switched communications system. The system breaks messages into fixed size packets and transmits the packets to the receiving Transputer. Packets are also referred to as blocks. The packet size can be set as the user wishes with the smallest size packet being 1024 bytes. For this Thesis, the packet size is set to 1024 bytes. If a message cannot be divided into full packets, the last packet will be smaller than a full packet. In other words, Express does not pass pad a message to send a full packet. [PAR 90]

The number of buffers available is also selectable by the user. Each buffer is the same size as a packet. Therefore, if a message is larger than one block, it will require more than one buffer to store it. While the number of buffers is set by the user, the memory needed for the buffers is taken from system memory, so assigning more buffers will reduce the amount of memory available to a node program to run. [PAR 90]

The “exwrite” function is non-blocking and sends a message from the calling node to the designated node. This function allows any number of bytes to be transferred and allows these bytes to be assigned a “type”. This type is an integer assignment that has no relation to standard data types. It is simply a tag that the system assigns to a message to distinguish it when it is received. The routing the message takes is provided by the Express kernel and the message is buffered by the kernel until that processors node process is ready to read it. [PAR 90]

Messages traverse one or more communications links when being sent from one node to another. The route between any two nodes is assigned when the system is configured. This routing is fixed so that only one predetermined route between two processors is used. This route is determined through a shortest path algorithm. The configuration tool examines all possible routes between processors and then picks the

shortest possible route. In some cases, there is more than one route with the shortest length. In this case, the kernel picks the route at random. [HIC 95]

The “exread” function, unlike “exwrite”, is a blocking function that receives a given number of bytes from another node. The function scans the message buffer that the Express kernel provides to determine if the node has received an appropriate message. A message is deemed appropriate if it came from the expected node and is of the right “type”. This is where the type of the message is significant. A message can be given a type so that it can only be read by a specific “exread” call. If there is more than one message that meets the sender and type requirements, the first message to arrive is chosen. The message length is insignificant in the selection of an appropriate message. If the message is too long, the extra bytes are discarded. If the message is too short, the read is completed with the smaller message. [PAR 90]

Since Transputers only provide blocking reads and writes, the Express kernel is designed to allow writes to occur even when there is no process ready to receive the message. Express does this through the use of daemons. Each Transputer has a daemon running for each link which looks for incoming messages and stores them in a buffer. When the running process executes a message read, the buffer is checked for the appropriate message. [HIC 95]

Sending and receiving messages between the host processor and node processors requires some extra considerations. While the same functions are used to send and receive messages, since the nodes are one type of processor and the host is another, type sizes and byte ordering must be considered. This is usually not a problem in node to node communications since the nodes are usually identical or similar processors.

Since different processors use different sizes for the various basic data types, this must be considered in the function call. If the host is sending a message that contains a certain data type that is represented by two bytes, and the node uses four bytes for that data type, the receiving processor will be expecting a different number of bytes than are sent. This could result in meaningless data being received.

Likewise, the order that bytes are stored can also cause problems. If the host processor uses big endian notation and the nodes use little endian notation, or vice versa, the bytes will be received in the wrong order. Express does take this into consideration and provides functions that perform byte swapping to rearrange a message either before it is sent or after it is received [PAR 90].

A non-blocking read can be established through the use of the “extest” function. This function looks in the message buffer for a message from a given node and of a specified type. If a suitable message is found, the function returns the length of the message found. If no suitable message is found, the function returns a negative value. By calling this function prior to any reads, it can be determined if a suitable message is currently in the buffer. If not, the program can continue execution and call “exread” at a later time. This is only of use, of course, if the data in the expected message is not immediately needed. [PAR 90]

d. Additional Communications Functions

Other communications functions are available that utilize the Express kernel. These provide broadcast communications, message concatenation, file reads and writes, in addition to other functions.

Express also provides communications functions which allow the user to route the messages along whichever Transputer port he desires. These functions, “exchanrd” and “exchanwt”, send a specified number of bytes along the specified Transputer link of the node. These functions do not provide message typing as the “exwrite” and “exread” functions do. These functions are also synchronous. This requires that the receiving node must be reading the appropriate channel while the sending node is writing to it. Since no routing is provided, the messages may only be sent to the nearest neighbor and that node is then responsible for passing the message on. [PAR 90]

e. Multitasking

Multitasking is also provided by the Express kernel. This is provided by a message handler function called “exhandle”. This function is a non-blocking read function. “Exhandle” is designed to scan the message buffer for a message from a designated node and of a designated type. If an appropriate message is found a function designated by the “exhandle” function is executed as a separate thread. If no appropriate message is found, the program continues execution as if the “exhandle” call never happened. Express also provides semaphores to allow the user to set up mutual exclusion for variables. [PAR 90]

3. Familiarization Testing

Several different test programs were written and run in order to determine the various characteristics of the Express message passing system and the Express kernel. Some tests only varied from previous tests in scale. That is, only the number of processors being utilized was changed.

a. Simple Message Passing

The first group of tests were performed in order to get an understanding of how the “exread” and “exwrite” functions execute. In the first test, a message was sent from the host to processor zero. This message contained an integer. It was then added to a constant and the result sent back to the host. The type of the message was set to a constant. This initial test was performed to ensure that the message would actually get to processor zero, be read correctly, and the correct result would be returned.

After this was determined to be working correctly, an additional processor was introduced. The test remained the same, except that the result of the addition was sent to processor one, where it was multiplied by a constant, and then returned to the host. This was performed to ensure that the processors of the system would properly send messages to one another.

Next, a third processor was introduced. This test had two processors receiving a message containing an integer from the host. These processors added a constant to the

received integer and both then sent it to the third processor where the two were multiplied together. The result was then sent back to the host. This tested the processor's ability to receive multiple messages simultaneously.

The problem was then increased to four and finally five processors to ensure that no problems would be encountered when adding more processors.

b. Changing Message Types

Once it was determined that extra processors could be added without complication, the message typing system was tested. The five processor test was again run, except this time the type of the message was set differently depending upon which processor the message was being sent to. Each node program would receive a message with type set to a specific value (100 for processor zero, 101 for processor one, etc.). It would then send messages to other processors using the receiving processors assigned type. This tested the ability to change types as the user desires and that message types could be used to distinguish between messages. This was attempted sending messages with incorrect types and these were not received.

c. Sending and Receiving Multiple Messages

Another test was performed with slightly more message passing to determine how complex a programs message passing could get. Messages were sent back and forth between nodes, and to multiple nodes. This had no effect on the reception of the messages as all messages were received properly. This leads to the conclusion that as long as the programmer takes care to ensure that functions to send and receive messages are properly used, message passing can be as simple or complex as the program needs.

d. Sending Messages Internally

The ability of a node to send a message to itself was tested. A message was sent from a node program to itself using the "exread" and "exwrite" functions. This worked in identical fashion to sending messages from one node to another. Since the PS heuristic

occasionally requires nodes processors to contain several node programs, this feature was tested to ensure that these instances would not require special handling.

e. Non-blocking Reads

The “extest” function was tested next. The three node program used before was used, except that the “exreads” in the multiplication were replaced by a loop with “extests”. This loop continuously tested the buffer for received messages. Once two were received, the loop terminated and the multiplication was performed. This tested the use of the “extest” function as a non-blocking read.

f. Multitasking

The message handler “exhandle” was tested to ensure proper functioning. A node program was written which had an addition and a multiplication function written. When a message was received of one type, the addition would be performed. If the message was of another type, the multiplication was performed. The program was run having the host send various messages to the node to ensure that the proper function was executed. This worked as expected.

g. Program Loading

The ability to load two node programs on one processor was also tested. It was attempted to load two programs on one node to see if both would run. The first attempt resulted in the program not properly executing. Since no output was received, the cause of the problem could not be determined. The program was run again, this time using two node programs that only received a message from the host and sent a unique message back to the host after receiving the message. This was tested by having the host send a message to each node program and seeing what was returned. The node program that was loaded last always was the only one to return a message. This indicates that the program that was loaded last is the only one that the node processor recognizes.

h. Program Looping

Two tests were run to check program looping. One tested using loops in the node programs and the other using loops in the host. When loops were used in the nodes, to simulate multiple execution of the program, as long as the loops were designed so that each node program was executed the same number of times as the others, the program executed properly. When the loop was moved to the host program, it was noted that the node programs had to be reloaded when their execution had completed. The node programs could not just be restarted. These tests indicate that for programs where multiple iterations of the program are needed, it is more efficient to have each node program use a loop than have the host program loop.

i. Message Reception Order

The last group of tests were performed to determine if the order of message reception had an effect on how the node program would read them. These tests utilized the two node program used earlier, except that the multiplication was modified to read a message from both the addition node and the host.

In the first test, the message from the host was delayed so that the message from the addition was ensured of being received first. The multiplication was written so that it tried to read the message from the host first. This program ran with no problem, indicating that the order that messages are received is unimportant.

Next, the multiplication node program was loaded, but not executed, until after the messages were sent to it. The node was then started. This also had no problems running. Finally, the multiplication node was not loaded, until after the messages were sent to it. This program did not work. This leads to the conclusion that as long as the node has been loaded, when the message is sent to the node is unimportant.

4. Communications Model

During performance testing, the main goal was to determine the effects of various message passing schemes on the Transputers performance. Many different tests were run

with each varying the test parameters slightly. An Express function, “extime”, was used to time the required sections of the programs execution. This function returns time in microseconds and is accurate to 64 microseconds. This is due to the clock tick that is used for time measurement occurs every 64 microseconds.

Since different processors store basic data types in different manners, the Transputer was checked to determine how many bytes were utilized in storing the basic data types. By using the “sizeof” function in the C language, it was found that characters are 1 byte, short integers, integers, long integers, and floating points are 4 bytes, and double precision floating points and long double precision floating points are 8 bytes. This data was needed in order to determine the length of messages that were passed between processors during test runs.

Due to the inherent inaccuracies of the timing functions, executing one message send or one message receive would be susceptible to inaccurate timing. In order to avoid this, all communications tests were conducted using a loop of 1000 iterations. The loop overhead was determined and found to be an average of 2060.8 microseconds. This was determined through 5 runs of a loop containing no function calls. All other test runs were run 5 times and the results averaged. The results were then adjusted for the loop overhead and divided by 1000 producing the time to execute one iteration of the test case.

a. Message Passing Testing

The first group of experiments was designed to test the message travel time of various sizes of messages across a varied number of links. The Transputer system that was used for these tests was an eight processor hypercube. In doing these tests it was desired to learn how the time was affected by increasing the length of the message and also by increasing the number of links traversed.

The programs that were designed for these experiments consisted of a sender node and a receiver node. The sender node consisted of a 1000 iteration loop that sent a message of a given size to the receiver node and then received the message back. The

receiver node consisted of a similar loop except that it listened for the message from the sender and when it received it, sent it back. The receiver node was placed on a particular processor so that the message would traverse a predetermined number of links. Since the messages traveled to the receiver and back, the total time was divided by two to yield the message send time. This is depicted in Figures 15, 16, and 17.

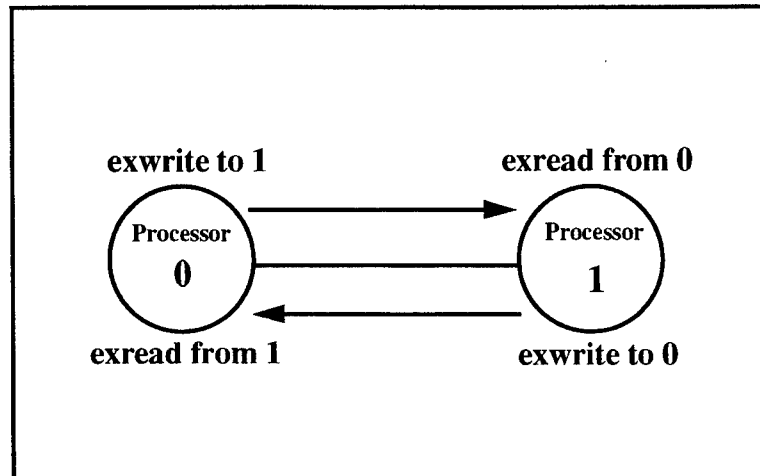


Figure 15: Message passing test for one link.

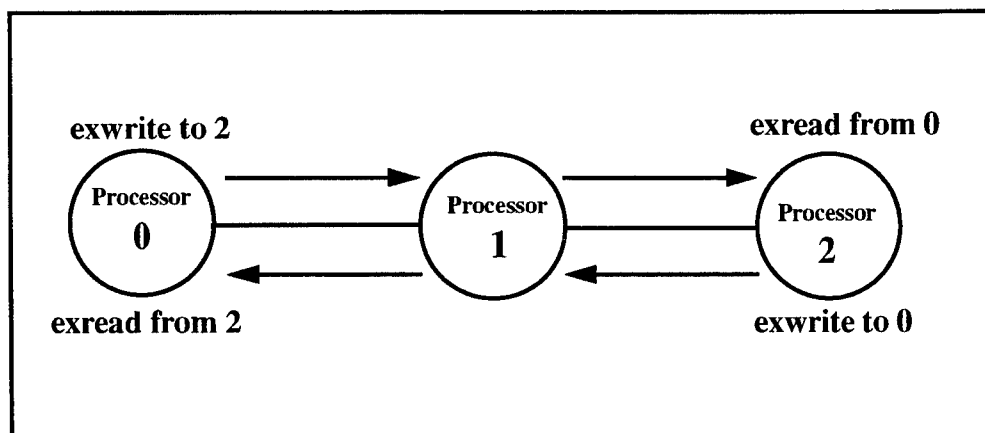


Figure 16: Message passing test for two links.

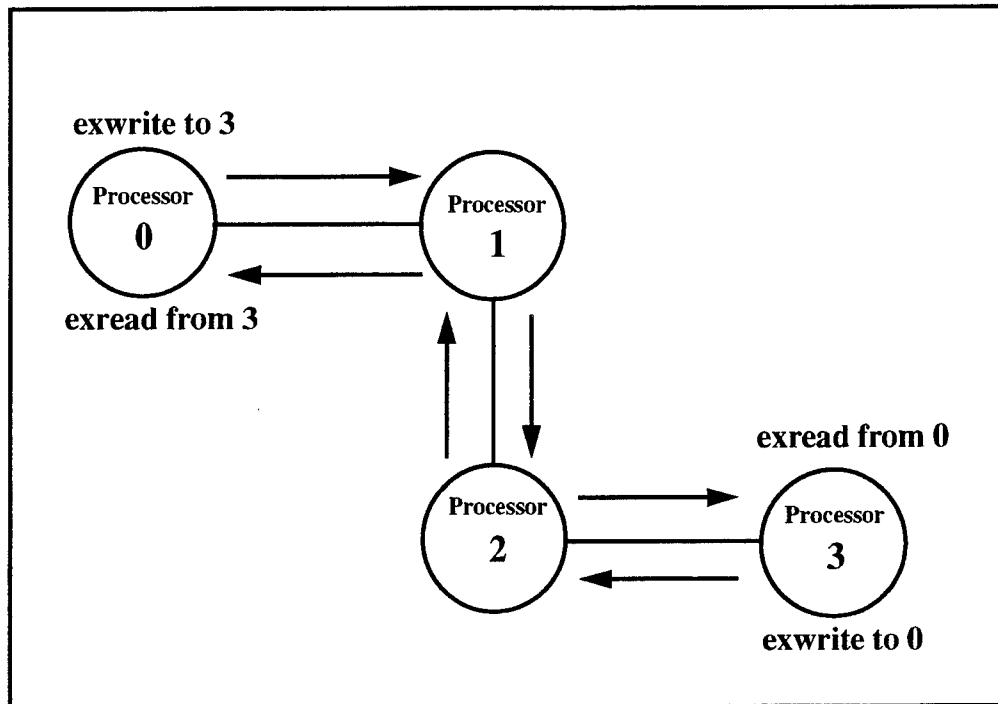


Figure 17: Message passing test for three links.

Many different message sizes were tested. These sizes were selected to determine time required to send a byte of data and determine the effect of sending more than one block of data. These were tested for traversal of one, two and three links. The resulting times are shown in Table 1.

Message passing was attempted using different message types with the same number of bytes. Arrays of characters, structures of different data types, arrays of integers, and arrays of floating points were setup to total the same number of bytes. The messages were passed and it was noted that the data types used had no effect on the resulting message passing time. The time was the same regardless of the data type.

As can be seen by looking at each column of the table, as the number of bytes in the message increases, the time for the message to traverse the communications path increases. In order to determine if there is a regular pattern to the time increase, an equation

can be generated to model the message sending process. The linearity of the communications times, as seen in Figure 18, suggests that a pattern exists.

Number of Bytes	1 Link Traversed	2 Links Traversed	3 Links Traversed
1 byte	303.6416	373.9776	445.2864
4 bytes	307.6352	382.0672	457.0176
8 bytes	312.3072	391.3408	470.656
40 bytes	353.1968	468.0128	582.8224
80 bytes	401.3696	559.3984	718.3552
160 bytes	449.4144	745.5169	992.5184
1000 bytes	1531.84	2701.824	3871.872
1024 bytes	1562.08	2758.08	3954.688
1025 bytes	1717.44	2914.4	4111.136
1536 bytes	2345.02	3975.81	5172.16
2048 bytes	2973.76	5168.192	6364.8
2049 bytes	3130.144	5324.48	6520.832
2560 bytes	3757.89	6388.67	7585.15
3072 bytes	4386.85	7581.44	8777.12
3073 bytes	4543.01	7737.44	8933.82
3584 bytes	5170.02	8801.54	9997.66
4096 bytes	5799.552	9993.568	11189.888
4097 bytes	5955.392	10150.016	11346.528

Table 1: Communications times (in microseconds).

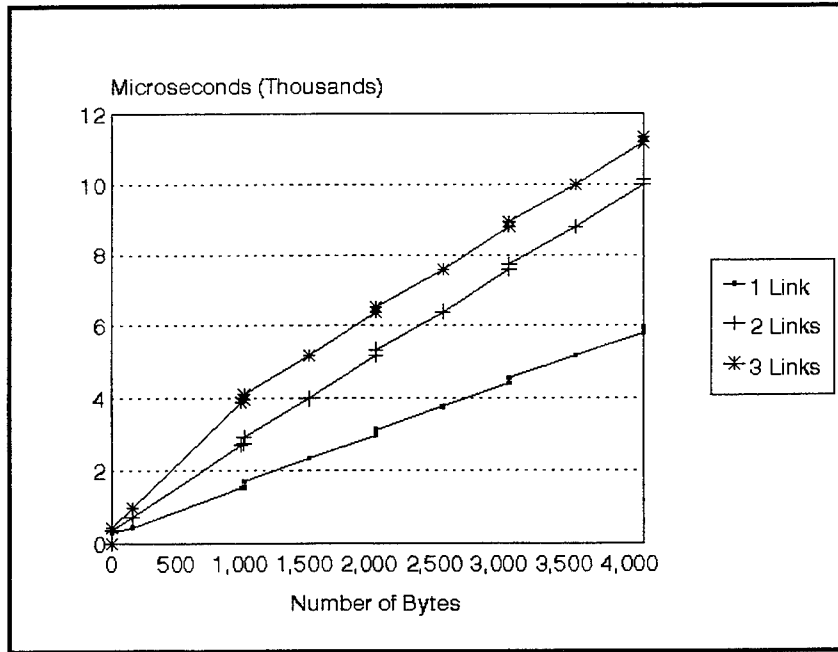


Figure 18: Communications times.

The pattern that arises consists of four basic parts. The overhead associated with the send command itself, the overhead per block, the overhead for each node the message passes through, and the transmission time of each byte. As can be seen from the graph, the byte transmission time takes on different values depending on what block it is associated with and how many nodes the message is traveling. A model for the system is given in Equation 1.

$$\begin{aligned}
 \text{Time} &= T_i + (T_{bl} \times \text{blocks}) + (T_{by1} \times \text{bytes}) && \text{for first link} \\
 &+ T_n + (T_{by2} \times \text{bytes}) && \text{for first block of 2nd link} \\
 &+ (T_{by3} \times \text{bytes}) && \text{for subsequent blocks of 2nd link} \\
 &+ T_n + (T_{by2} \times \text{bytes}) && \text{for each addl. link, max. 1 block}
 \end{aligned} \tag{Eq 1}$$

$$\begin{aligned}
 T_i &= 150\mu s & T_{bl} &= 155\mu s & T_n &= 70\mu s \\
 T_{by1} &= 1.227\mu s & T_{by2} &= 1.1\mu s & T_{by3} &= 0.977\mu s
 \end{aligned}$$

T_i represents the overhead due to the instruction call. This is overhead that is incurred to simply make a message read or write function call. It is an overhead that is

associated with every message sent. T_{bl} is the overhead associated with each block of the message. T_n is the overhead associated with every node the message passes through.

The byte transmission times are T_{by1} , T_{by2} , and T_{by3} . Even though transmission of a byte should take the same length of time no matter what link it travels on, the times observed yielded three different times depending on which block of the message was being transmitted and whether it was the first, second, or a subsequent link that the block was traversing. This is most likely due to pipelining in the system which allows some overlapping of the per byte time.

Equation 1 shows that there are five different cases to be considered in message passing. These are messages traversing one link, single block messages traversing multiple links, multiple block messages traversing two links, single block messages traversing more than two links, and multiple block messages traversing more than two links.

For a message traversing one link, the first part of Equation 1 is used. Suppose a message of 2048 bytes traversed one link. The resulting message passing time would be found as in Equation 2.

$$\begin{aligned}
 \text{Time} &= T_i + (T_{bl} \times \text{blocks}) + (T_{by1} \times \text{bytes}) \\
 &= 150 + (155 \times 2) + (1.227 \times 2048) \\
 &= 2972.896\mu s
 \end{aligned}
 \tag{Eq 2}$$

Equation 3 describes the calculation of a message of 1000 bytes traversing two links. This is an example of the second case, a single block message traversing multiple links. The second line of Equation 1 is needed in this calculation.

$$\begin{aligned}
 \text{Time} &= T_i + (T_{bl} \times \text{blocks}) + (T_{by1} \times \text{bytes}) \\
 &\quad + T_n + (T_{by2} \times \text{bytes}) \\
 &= 150 + (155 \times 1) + (1.227 \times 1000) + 70 + (1.1 \times 1000) \\
 &= 2702\mu s
 \end{aligned}
 \tag{Eq 3}$$

For the third case, multiple block messages traversing two links, the third part of Equation 1 is needed. Equation 4 shows the calculation of a 2048 byte message traversing two links.

$$\begin{aligned}
\text{Time} &= T_i + (T_{bl} \times \text{blocks}) + (T_{by1} \times \text{bytes}) \\
&\quad + T_n + (T_{by2} \times \text{bytes}(\text{first block})) \\
&\quad + T_{by3} \times \text{bytes}(\text{addl. blocks})
\end{aligned} \tag{Eq 4}$$

$$\begin{aligned}
&= 150 + (155 \times 2) + (1.227 \times 2048) + 70 + (1.1 \times 1024) + (0.977 \times 1024) \\
&= 5169.744\mu s
\end{aligned}$$

Case four is single block messages traversing more than two links. The example given is a 1000 byte message traversing three links. The fourth line of Equation 1 is used for this case. Equation 5 shows this calculation.

$$\begin{aligned}
\text{Time} &= T_i + (T_{bl} \times \text{blocks}) + (T_{by1} \times \text{bytes}) \\
&\quad + T_n + (T_{by2} \times \text{bytes}) \\
&\quad + T_n + (T_{by2} \times \text{bytes})
\end{aligned} \tag{Eq 5}$$

$$\begin{aligned}
&= 150 + (155 \times 1) + (1.227 \times 1000) + 70 + (1.1 \times 1000) + 70 + (1.1 \times 1000) \\
&= 3872\mu s
\end{aligned}$$

The fifth case is multiple block messages traversing more than two links. This case uses the restriction placed on the fourth line of Equation 1 that only the first block of the message figures into the message passing time for links traversed after the first two. The calculation for a 2048 byte message traversing three links is given in Equation 6.

$$\begin{aligned}
\text{Time} &= T_i + (T_{bl} \times \text{blocks}) + (T_{by1} \times \text{bytes}) \\
&\quad + T_n + (T_{by2} \times \text{bytes}(\text{first block})) \\
&\quad + T_{by3} \times \text{bytes}(\text{addl. blocks}) \\
&\quad + T_n + (T_{by2} \times \text{bytes}(\text{first block}))
\end{aligned} \tag{Eq 6}$$

$$\begin{aligned}
&= 150 + (155 \times 2) + (1.227 \times 2048) + 70 + (1.1 \times 1024) \\
&\quad + (0.977 \times 1024) + 70 + (1.1 \times 1024) \\
&= 6366.144\mu s
\end{aligned}$$

Finally, if a message needed to travel over more than two links, only a maximum of one block of data added additional per byte time. This is also most likely caused by pipelining. In this case, different blocks of the message can be traversing different links of the route at the same time. This would cause the results that were observed.

One other related test was performed to examine the affect of breaking a message into parts and sending each part individually. For this test, the forty byte message

was broken into four ten byte messages and each of the ten parts were sent one after another. The resulting average time was 2389.3696 microseconds when traversing one link. When comparing this to sending the message whole, it is quite obvious that splitting the message into parts is much worse due to a large increase in overhead.

b. Internal Message Passing

Since the PS implementation may need multiple tasks to be mapped to one processor, message passing from a node to itself was tested. The same send and receive loop was again used except the receive function was set up to receive a message from itself. The resulting times are shown in the Table 2.

Number of bytes	Time
1 Byte	369.0752
4 Bytes	368.8576
8 Bytes	370.1632
40 Bytes	378.1248
80 Bytes	389.4912
160 Bytes	409.792

Table 2: Communications times for internal message passing (in microseconds).

It can be seen from the table, increasing the size of the message increases the time slightly. This time does not increase in proportion to T_b found before since the message does not traverse any links. Also, comparing these times to the message times found previously seems to indicate that it takes longer for a message to be sent to itself than to send it to another node in some cases. This is most likely caused by the test loop. Since the “exread” function cannot be called until the “exwrite” has completed since they are part of the same program, this most likely accounts for the longer times observed.

Since no links are used when a Transputer passes a message to itself, it would seem that passing the message internally would take far less time than passing it to another Transputer. The data shows that this is not the case. The most likely cause for this is time required to execute the Express read and write functions, overhead time for preparing the message, and time used to search the buffer.

c. Routing Delays

The last set of experiments tested the affect of messages passing through a node that has a process currently running. These tests attempted to determine if the running process, the message being sent, or both were affected.

A message of 160 bytes was sent in the same fashion as before to a node such that it passed through one and two nodes. These nodes contained a process that consisted of an infinite loop. The average times observed for the two tests were 747.2064 microseconds and 995.6224 microseconds respectively. When compared with previous results, it is shown that the running process had little to no affect on the message propagation time. Figure 19 shows a depiction of this test.

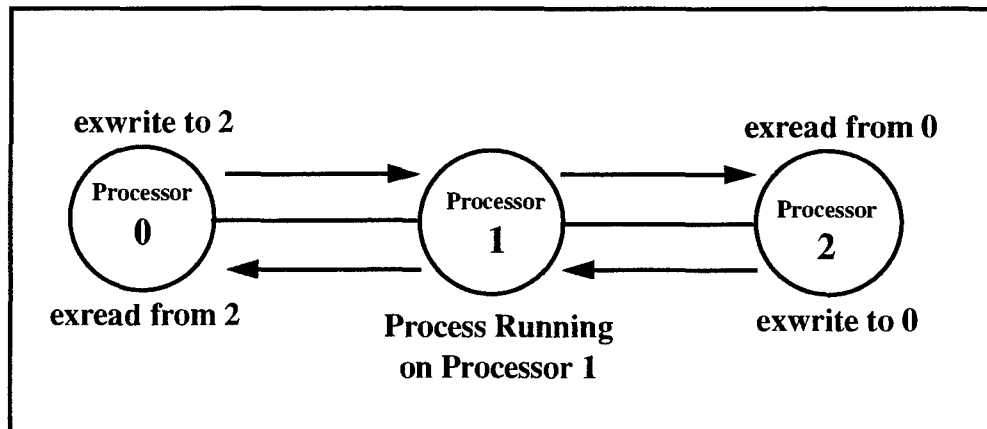


Figure 19: Test for effect of routing on computation.

Next, the affect on the running process was examined. A process that consisted of a loop that executed an addition operation was timed with no messages passing through

the node and then timed again with a message passing through. The execution times are shown in Table 3.

Number of Messages	4 byte messages	40 byte messages	1024 byte messages	1025 byte messages	2048 byte messages	2049 byte messages
0	560.896	560.896	560.896	560.896	560.896	560.896
10	561.920	561.984	563.968	565.056	567.040	568.192
20	563.008	563.136	567.040	569.216	573.312	575.424
30	564.160	564.352	570.176	573.440	579.456	582.720
40	565.248	565.504	573.248	577.600	585.728	590.016
50	566.336	566.720	576.384	581.824	591.872	597.312

Table 3: Process running times with messages passing through (in milliseconds).

By the times recorded, several observations can be made. Messages passing through a node have an obvious effect on the process currently executing. The running time of the process increased when messages pass through. Also, the increase in time is affected by both number of messages passing through and the number of bytes, but not the number of blocks the message is divided into.

By analyzing the numbers in Table 3, a pattern for the delay caused by messages passing through a node is seen. Each message carries an associated delay with it in addition to an additional delay for each byte the message contains. This is shown in Equation 7 where T_{bl} is the delay per message and T_{by} is the delay per byte.

$$\text{Delay} = T_{bl} + (T_{by} \times \text{bytes}) \quad (\text{Eq 7})$$

$$T_{bl} = 54\mu s \quad T_{by} = 0.1\mu s$$

Finally, the effects on communications for a node with messages passing through was studied. A message passing loop was run on two processors with another

message passing loop passing messages through one of the executing nodes. This is shown in Figure 20.

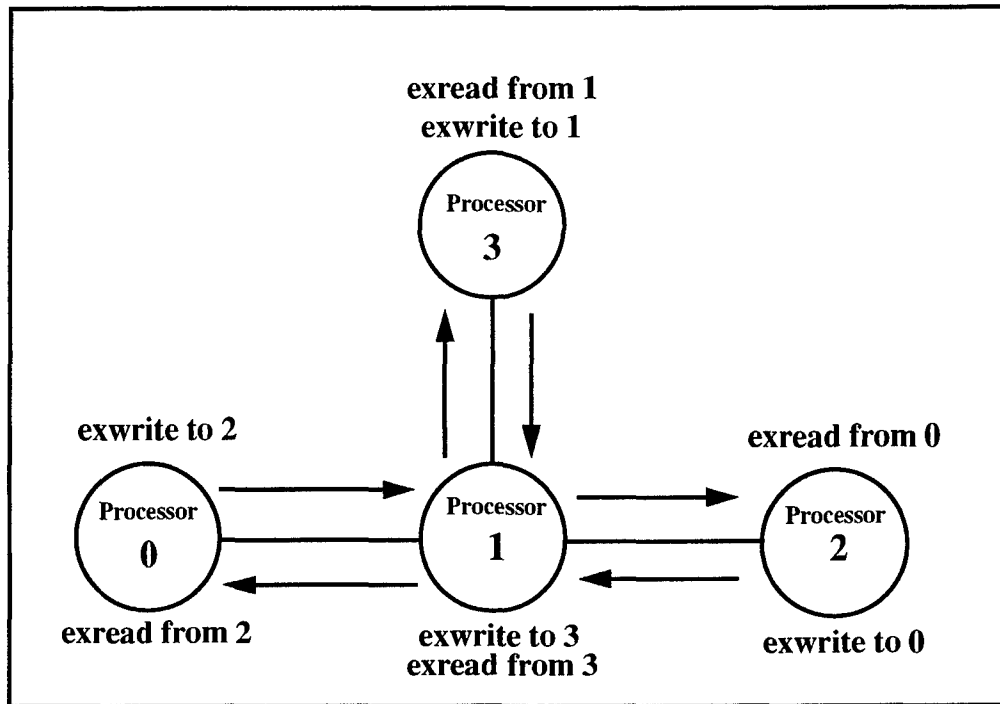


Figure 20: Test for effect of routing on communication.

The loop was timed first with no messages and then with a variety of messages passing through. The results are shown in Table 4.

These results indicate that communications are affected by message forwarding in the node. Furthermore, they show that the number of messages has a greater affect than the number of bytes in the message. These results are similar to the ones found in the previous experiment.

These findings lead to the conclusion that message forwarding takes the highest priority of all functions executed by the Express kernel. Also, communications seem to be affected slightly less than computation leading to the conclusion that communications is higher priority than computation.

1000 Iteration Communications Loop	Average Total Time
No passing messages	707.085ms
50 messages, each 200 bytes	709.709ms
2500 messages, each 4 bytes	741.350ms
150 messages, each 200 bytes	715.059ms
150 messages, each 4 bytes	712.832ms

Table 4: 1000 iteration communications loop times.

IV. RPS HEURISTIC AND PROGRAMMING TOOLS

The Realistic Periodic Scheduling (RPS) environment is a system which schedules a process according to the RPS heuristic developed. It interprets the code written by the programmer and composes source code that, when compiled, will execute the process according to the generated schedule. This system requires the programmer to generate a task graph and write source code for the nodes using the C programming language using annotations to describe message read and write operations. The RPS scheduler and RPS packager are both contained in one executable which schedules the program and then packages the code. An additional tool, the RPS profiler, is also provided which profiles code written to determine actual execution time of the code.¹

A. RPS SCHEDULER

The scheduling portion of the system implements the RPS heuristic, which takes into account various aspects of the Transputer/Express system. The programmer must provide several information files, in proper format, for the system to properly schedule the process on the Transputers. The RPS scheduler generates a schedule file that is used by the RPS packager to generate source code and may also be used by the programmer to analyze the schedule that has been assigned.

1. Transputer/Express System Model

There are several factors to consider when scheduling nodes to processors using Inmos Transputers and the Parasoft Express software package. The PS heuristic considers communications, resource contention, and the underlying interconnection network of the system in determining the resulting schedule. It does not, however, consider effects of message setup, processing overhead, and message transit time on processor utilization, message forwarding by a processor, and actual routing paths utilized by the system. Each

1. This tool does not give communication time since the system is designed using the communication model shown in Chapter III.

of these has an effect, that cannot be ignored, on the execution time of the nodes of the task graph. Additionally, input and output between the system and the outside world have various effects on the system which can affect both the schedule or the overall execution time of the program.

a. The Communications Model

Message setup time and execution time of message passing commands go hand in hand when determining schedule length. In PS, each nodes execution time is determined strictly by the computation time associated with that node. In the Express system, message passing commands incur an overhead due to message setup (or breakdown on reads) and command execution time. The RPS scheduler considers this time when determining the schedule as the processor is not free to execute another node until these actions are completed. The effect of this overhead time essentially increases the nodes execution time by the sum of overhead for each read and write command used by the node.

In addition, the processor is utilized during the actual transmission of the message. Since the Transputers in this system do not have a separate communications processor, each message that is sent or received requires the use of the processor to accomplish the message passing. This means that the sending processor will be unable to begin computation for the next node until the message transmissions are completed.

It is assumed that once all blocks have been sent to the next processor, the sending processor can continue computation. It appears that a processor sends one block to the next, that block is forwarded, then the next block is sent. This means that unless only one block is being sent, each block must transit 2 links before the sending processor can send the next block. Because of this, the RPS scheduler was modeled assuming that once each block had transited two links, the processor was free to continue computation.

b. Routing Overhead Model

The forwarding of messages by a processor in a system using Parasoft Express has an effect on the execution time of the process that is executing on the processor. If a

message is received by a processor which is to be forwarded to another processor, the executing process is forced to share execution with the message forwarding operation. This results in the executing node having a longer execution time than was originally determined.

The overall result of this characteristic of the system is that processors which are used to forward messages will have a longer execution time for one revolution of the cylinder than originally planned. Since it cannot be known before hand whether this increase in time will result in a less efficient schedule or not, the RPS scheduler examines the effect of message forwarding on the overall schedule. Scheduling a node on a processor which will require no message forwarding, but a slightly longer execution time might be more efficient depending on the additional time required by message forwarding. Therefore, this factor is taken into account by RPS.

c. Determining Routing Paths

The PS heuristic schedules the tasks by using shortest path routing. This makes sense since a static routing scheme is most likely to use a shortest path routing algorithm for message passing. The difficulty arises when there is more than one shortest path. For example, in a four processor ring, the opposite processors in the ring have 2 two link paths to each other. PS has no consideration for which path will be taken.

Express uses a shortest path static routing system to determine the path taken by messages [HIC 95]. This routing is established when the system configuration is defined. The paths are fixed prior to loading the processes onto the processors and does not change throughout execution. Because of this, only one of the shortest paths is utilized and it is predetermined. Since this is known ahead of time, the RPS heuristic considers only the actual routes that are used by the Transputer network during process execution.

d. Input and Output Effects

The Express system uses a host PC to load the Transputers with the code they are to execute. The host PC is also the only processor that can receive input or produce

output for either standard I/O or files. Because of this, any I/O that needs to be done by a process must be passed as a message to the host PC which will then perform the required I/O.

This limitation adds an additional consideration to the schedule. Any node that generates data for output or requires input will achieve this through message passing with the host. If the processor is directly connected to the host, this will have no effect on the schedule, except for message setup and command overhead, as the links between the host and the Transputers are not utilized for passing messages between Transputers in the system. However, should a Transputer not be directly connected to the host, the message must be routed through other Transputers before arriving at the host. This will have an effect on the schedule for the reasons discussed in the previous sections.

One additional factor with I/O does not relate to scheduling: sampling time. Should the host PC perform the I/O for one iteration of the cylinder slower than the slowest processor in the Transputer network, the overall execution time of the process will be determined by the host PC. This cannot be improved by scheduling the graph differently on the Transputer network.

2. Input to the Scheduler

The RPS scheduler requires two kinds of information to determine a schedule. First it needs the routing paths for the system. Second it needs the task graph description. This information is stored in two files: `routing.cgd` and `graph.cgd`.

The first line in the routing file contains the number of processors. The rest of the lines contain information about all combinations of source and destination processors. Each successive line lists the source processor, the destination processor, the number of links that must be traversed between the two, and then the list of intermediate processors. When constructing `routing.cgd`, no path information for same processor message passing should be included. A path must be listed for each processor pair or an error will occur. The path from any processor to the host PC is not listed. It is determined from the path to processor

zero since the host PC is connected to processor zero. The routing.cgd file for a 4 processor hypercube is given in Figure 21.

```

4
0 1 1
0 2 2 1
0 3 1
1 0 1
1 2 1
1 3 2 0
2 0 2 1
2 1 1
2 3 1
3 0 1
3 1 2 0
3 2 1

```

Figure 21: Routing.cgd file for a 4 processor hypercube.

The format for the graph is that the first line contains the number of total nodes in the graph followed by the number of total edges. A blank line is next followed by the node information. Each node is listed in the form shown in Figure 22.

Node Number					
Node File Name (without extensions)					
Node Function Call Name					
Computation Time					
Number of Incoming Messages	Parent 1	# of bytes	Parent n	# of bytes
Number of Outgoing Messages	Child 1	# of bytes	Child n	# of bytes

Figure 22: Node description format.

Each of these node descriptions is separated by a blank line. The host node function is numbered zero and each other node is numbered according to the programmers choice. The nodes must be listed in numerical order in the graph file or the program will not operate properly. The host node is used for I/O or any other functions that must be performed on the host. The routing file must be named routing.cgd and the graph must be named graph.cgd.

The RPS scheduler will output a file named `schedule.cgd`. This file is used by the RPS packager to generate the Transputer code needed to run the program according to the schedule. This file is in proper format with additional information not needed for code generation listed at the end. This file provides the processor each node is assigned to, the order of execution, and the index associated with each node. These are needed for code packaging. In addition, overhead time and computation time for each node, routing delay time, expected execution time, sequential execution time, and expected speedup are also listed. This way, the schedule file can be read by the programmer to see what the RPS packager has produced.

3. RPS Details

a. Scheduling Order

The RPS scheduler orders nodes according to some priority. The priority is determined by a combination of computation time plus the overhead associated with each message (assuming it traverses one link). This list is then used to schedule all of the nodes.

For example, suppose node N1 has a computation time of $600\mu\text{s}$ and sends one message of 10 bytes, and node N2 has a computation time of $500\mu\text{s}$ and sends two messages of 10 bytes. Sending 10 bytes of data carries an overhead of $317\mu\text{s}$ when traversing one link. This means that the total time for node one is $917\mu\text{s}$ and node two is $1134\mu\text{s}$. This means that node N2 would be scheduled first.

b. Processor Assignment

The RPS scheduler uses a modified version of the PS heuristic to determine the schedule. It includes overhead due to communications in determining the schedule. This is because the Transputer system requires that the processor perform the communications functions to send the message. These include message setup, breakdown, transmission, and reception. Also included in the scheduling process is instruction call overhead and delays in computation due to routing of messages through processors.

The RPS scheduler establishes a processor load table and a link load table as in PS and after attempting to schedule a node on each processor, the processor with the lowest maximum utilization is selected. The entries into the tables are made in a slightly different manner.

The computation time is added to the assigned processor. The communication time has to be computed by the RPS scheduler based on the number of bytes in the message. Each message write adds time to both the computation time and the communication time. This is due to the overhead discussed earlier. Because we assume that the processor is being utilized up until the time that all blocks of the message have been sent, this time is added to the processor load table for the assigned processor. Since each block must traverse two links before the next block can be sent, the processor is occupied with sending the message for the time it takes for each block except the last one to traverse two links and for the last block to traverse one. This time is added to the processors utilization.

The communication time associated with this message write also must be added to the link load table. Since a link is utilized when a block is traversing it, the time for each block to traverse a link is added to the link load table for each link in the path that the message takes.

During message forwarding, the daemon for the receiving link is utilized [HIC 95]. This means that the process running on the forwarding processor is timesharing the processor with the daemon. This adds time to the processor utilization of the forwarding processor which is characterized by Eq. 7 in Chapter III. This routing delay time is added to the forwarding processors utilization.

Receiving messages also adds time to the receiving processors utilization. Since the daemon is responsible for receiving all messages, it is assumed that the time to receive a message is the same as the time to forward it. This is added to the receiving processors utilization.

Communications between a Transputer and the host PC are modeled in the same way. The only difference is that the host PC and the links connecting it to the system are not considered when determining the lowest maximum resource utilization.

Figure 23 shows the pseudocode for this algorithm. This procedure establishes the Processor Load Table and Link Load Table values that reflect the ready task being assigned to a particular processor.

Two variations of the basic heuristic are considered. The first, called the simple heuristic, uses RPS to find the processor with the lowest maximum resource utilization. If there is more than one assignment that will produce this lowest maximum utilization, the node is assigned to some processor non-deterministically, in our implementation, it will be assigned to the lowest processor number. The other variation, called the complex heuristic, does the same thing except if more than one processor gives the lowest maximum utilization, the next highest utilization for these processors is examined. The lowest of these utilizations is the processor assigned. If there are still more than one processor, the process continues examining the next highest, until the tie is broken. If two processors result in identical utilization, the first processor attempted is the one assigned.

Figures 24, 25, and 26 show an example of this. Figure 24 shows the processor and link load tables before scheduling a particular node. Figure 25 shows the result of scheduling it on processor 2 and Figure 26 shows the result of scheduling it on processor 3. As can be seen, the maximum utilization for either case is 500.

Using the simple heuristic, the processor that would be picked is processor 3. This is because the first processor that found that yields the lowest maximum utilization is picked. Since the processors are tried in numerical order, processor 3 would be tried before processor 4 and thus be the first one found.

If the complex heuristic is used, the tie is broken by using the second highest utilization and taking the processor with the lowest value for it. In this case, both have a second highest utilization of 400. Continuing we find that the fifth highest utilization for processor 3 is 100 and processor 4 is 200. Therefore, processor 4 is picked.

procedure Schedule (Ready Task, Processor Load Table, Link Load Table)

for each Processor in the system

 add Ready Task computation time to associated
 Processor Load Table entry

for all predecessors of the Ready Task that have been scheduled

if on the same processor as the Ready Task **then**

 add buffer access time to Processor Load Table entry

else

 add message send time to associated Processor Load
 Table entry

 add message link traversal time Link Load Table for each
 link in route

 add routing delay time to Processor Load Table for each
 processor in route

 add message reception time to destination Processor Load
 Table entry

end (if)

end (for)

for all predecessors of the Ready Task that have been scheduled

if on the same processor as the Ready Task **then**

 add buffer access time to Processor Load Table entry

else

 add message send time to associated Processor Load
 Table entry

 add message link traversal time Link Load Table for each
 link in route

 add routing delay time to Processor Load Table for each
 processor in route

 add message reception time to destination Processor Load
 Table entry

end (if)

end (for)

end (for)

end.

Figure 23: Psuedo code for scheduling function of RPS.

P1	P2	P3	P4
500	400	0	0

Processor Load Table

P1	P2	P3	P4
---	100	0	---
0	---	---	0
0	---	---	0
---	0	0	---

Link Load Table

Figure 24: Example of load tables before node assignment.

P1	P2	P3	P4
500	400	300	0

Processor Load Table

P1	P2	P3	P4
---	100	200	---
100	---	---	0
0	---	---	0
---	0	0	---

Link Load Table

Figure 25: Load tables with node assigned to Processor 3.

	P1	P2	P3	P4
	500	400	0	300
Processor Load Table				

	P1	P2	P3	P4
P1	---	200	0	---
P2	0	---	---	200
P3	0	---	---	0
P4	---	0	0	---
Link Load Table				

Figure 26: Load tables with node assigned to Processor 4.

c. Index Assignment

Nodes belonging to different instances of the task graph can be concurrently executed. Hence, some index is required to identify the instance to which a node belongs. Index assignment is done recursively by the RPS scheduler. Starting with the first node in the graph, an index of zero is assigned. Then the *assign index* function is run for all parents and children of the node. Because of the recursive nature of the algorithm, the parents and children of each node are assigned an index. The algorithm used is the one presented by Bell [BEL 92]. A minor modification is made to handle parents and children assigned to the same processor since, in this case, no message passing time is needed. In this case, the parent and child can be assigned the same index if the parent is executed before the child in the cylinder iteration. If not, the parent is assigned an index of one higher than the child. Figure 27 shows an example of index assignments for a task graph segment. Since task A follows B in the schedule, task B must have an index of at least one greater than task A.

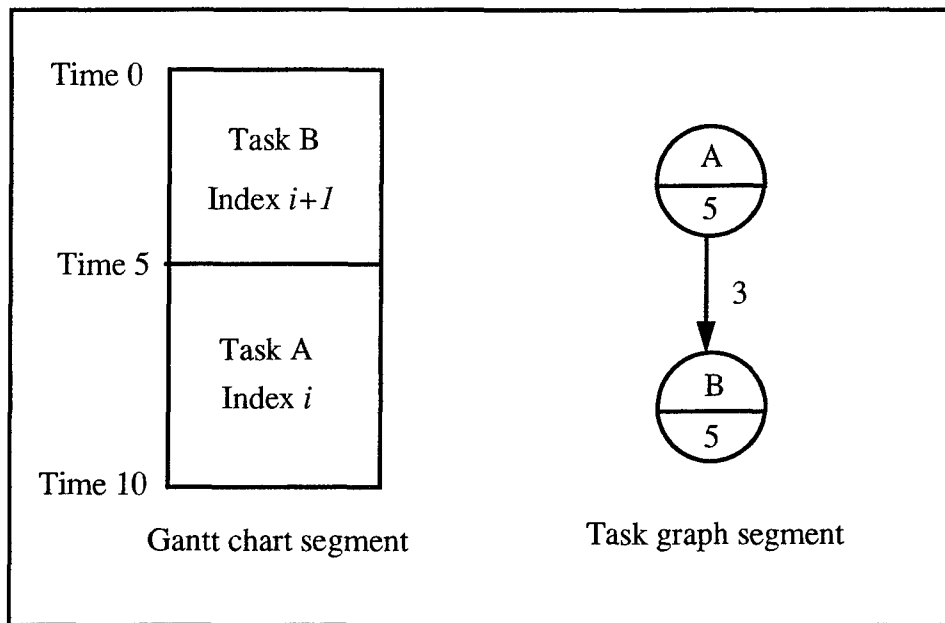


Figure 27: Example showing index assignments for segment of a task graph.

When determining the starting and finishing times of the nodes, two factors are considered. First, a node can not be considered finished until the message that is being sent to the child has arrived. Second, routing delays due to message passing affect the starting and finishing times of the node.

Since it cannot be determined exactly how the routing delay will affect the starting and finishing times, the RPS scheduler assumes worst case. The starting time of each node is figured assuming that the routing delay does not affect the starting time at all. This way, the earliest possible starting time of the node is used. For the finishing time, just the opposite is used. The finishing time is figured assuming the the entire routing delay occurs prior to the completion of the node. This way, the latest possible finishing time is figured.

A side effect of using this method is that the indices generated will take on both positive and negative numbers. The RPS packager is written in a way that only positive

numbers can be used. In order to accommodate this, the indices are adjusted so that the smallest index is set to zero and all other indices are adjusted accordingly.

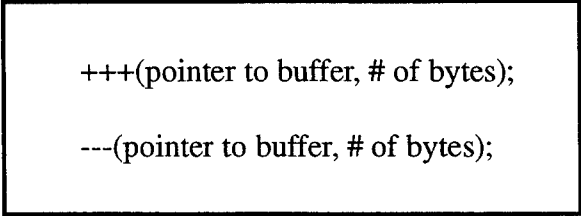
B. RPS PACKAGER

The RPS packager is a program which takes a task graph, a schedule for that graph, and code segments for each node in the graph and produces compilable source code which will execute on a Transputer network using the Parasoft Express system. This program also generates a batch program, compile.bat, which, when executed, will compile all source code produce executable code.

1. Read and Write Commands

In order to write the code segments for the graph nodes, message passing commands must be used to pass data between the nodes. Parasoft Express has commands for passing messages, but these require the programmer to know what processor the sender or receiver are located on. Since this is not known until after the schedule has been determined, special functions were devised that allow the programmer to write the code segments without knowing the location of sending or receiving nodes.

Reads and writes are written using these special annotations. They are the “+++” annotation to receive messages from other nodes and the “---” annotation to send messages to other nodes. These annotations must be written in the form shown in Figure 28.



```
+++(pointer to buffer, # of bytes);  
---(pointer to buffer, # of bytes);
```

Figure 28: Write and read annotations.

The pointer to buffer parameter is a pointer to the variable that holds the data to be sent or the location to store the data received. The number of bytes is the number of bytes that are to be sent in the message. The `sizeof()` function may be used for this parameter.

These annotations are converted into appropriate "exwrite" and "exread" commands by the RPS packager. Even though "exwrite" requires a destination processor and "exread" requires a source processor, this is not a parameter of the new annotations. The RPS packager uses the schedule file and graph file to determine the sources and destination nodes and the processors they have been assigned to. It then adds this to the packaged code.

Both "exwrite" and "exread" require a type to be assigned to the message. In order for a node to distinguish between two messages arriving from the same processor, these messages must have different types. In order to ensure that each message is distinguishable, each communications edge in the graph is assigned a number and this number is used as the type for the associated message. This way, each message has a unique type. These numbers are generated and added to the packaged code automatically by the RPS packager.

2. Code Generation Information Files

The RPS packager requires the two files that are required for scheduling, plus the output schedule file. It also requires the node code files and host code files.

Each node consists of three files, a .tcs file, a .inc file, and a .to file. The .tcs file contains the nodes code written in C using the two special annotations for message passing. The .inc file is written in the following format. The first line states the number of header files required by the node for compilation. Each subsequent line contains the header file that will be added to the processor code in the appropriate place during code generation. The file name should be enclosed in quotes or arrows as required by the C language. The .to file lists on each line the data type of the message that is to be sent by the node for all messages. If an array data type is used, it is listed using the data type and number of

elements in brackets (for example, int[15]). These must be listed in the order that the node expects the messages. Examples of .to and .inc files for nodes are given in Figure 29.

1 "node.h"	float int[10] struct nodestruct
---------------	---------------------------------------

Figure 29: Examples of .to and .inc files for nodes.

The host node requires three files. These are a .tcs file and a .inc file that are written in a similar manner as the Transputer node files. A .fil file is also needed which declares the files that are used for input and output. The first line of the file states the number of input files used. Each of the next lines lists each input file name. The next line states the number of output files with the subsequent lines listing the output file names. There is no .to file for the host node. An example of a .fil file is given in Figure 30.

1 input.txt 1 output.txt

Figure 30: Example of .fil file.

3. Generated Node Code

The RPS packager initially takes the graph and schedule files and reads them into memory. These files are labeled graph.cgd and schedule.cgd. Once the graph and schedule data have been read in, the RPS packager generates a compilable C code file for each processor in the system. This is accomplished by reading the node code for each processors nodes from .tcs files. These .tcs files are files which contain an individual node process.

The code that is generated for the Transputers takes the nodes that are assigned to each Transputer and creates a process that executes multiple iterations of the task graph. This is done using the revolving cylinder method. A loop is generated that executes each node

assigned to the processor during a loop iteration. The number of loop iterations to be made is also included in the code packaging. The code for each processor includes a function call to receive a broadcast message from the host that contains the desired number of iterations of the graph.

a. Flow of Execution

When creating the code for each processor, the RPS packager first determines which nodes are assigned to that processor by examining the schedule file. Once that has been determined, the .inc file for each node is read and the appropriate #include statements are placed into the code file. Global array variables are then added to the file which indicate the processor each node is mapped to and the type of each message.

The packager then examines the nodes mapped to the processor that code is currently being generated for. If there are messages passed between any two nodes on the processor, global buffer variables are included in the code for these messages.

The packager then puts copies of each nodes code into the code file. This is done by reading each line of the nodes .tcs file. After reading the line, it is examined to determine if it is a read or write. If it is a read, an "exread" command is added to the code file by determining the source processor and type of the message and adding this to the "exread" using the mapping and types variables. Likewise, if it is a write, an "exwrite" command is added to the code file by determining the destination processor and proceeding in the same manner as a read. If it is neither, the line is simply written as is into the code file.

The main procedure of the code is generated next. This includes local variables, and functions that control the number of iterations of the program. The main iteration loop is generated based on the indices determined by the revolving cylinder.

The flow of execution for creating Transputer code by the RPS packager is illustrated in Figures 31 and 32.

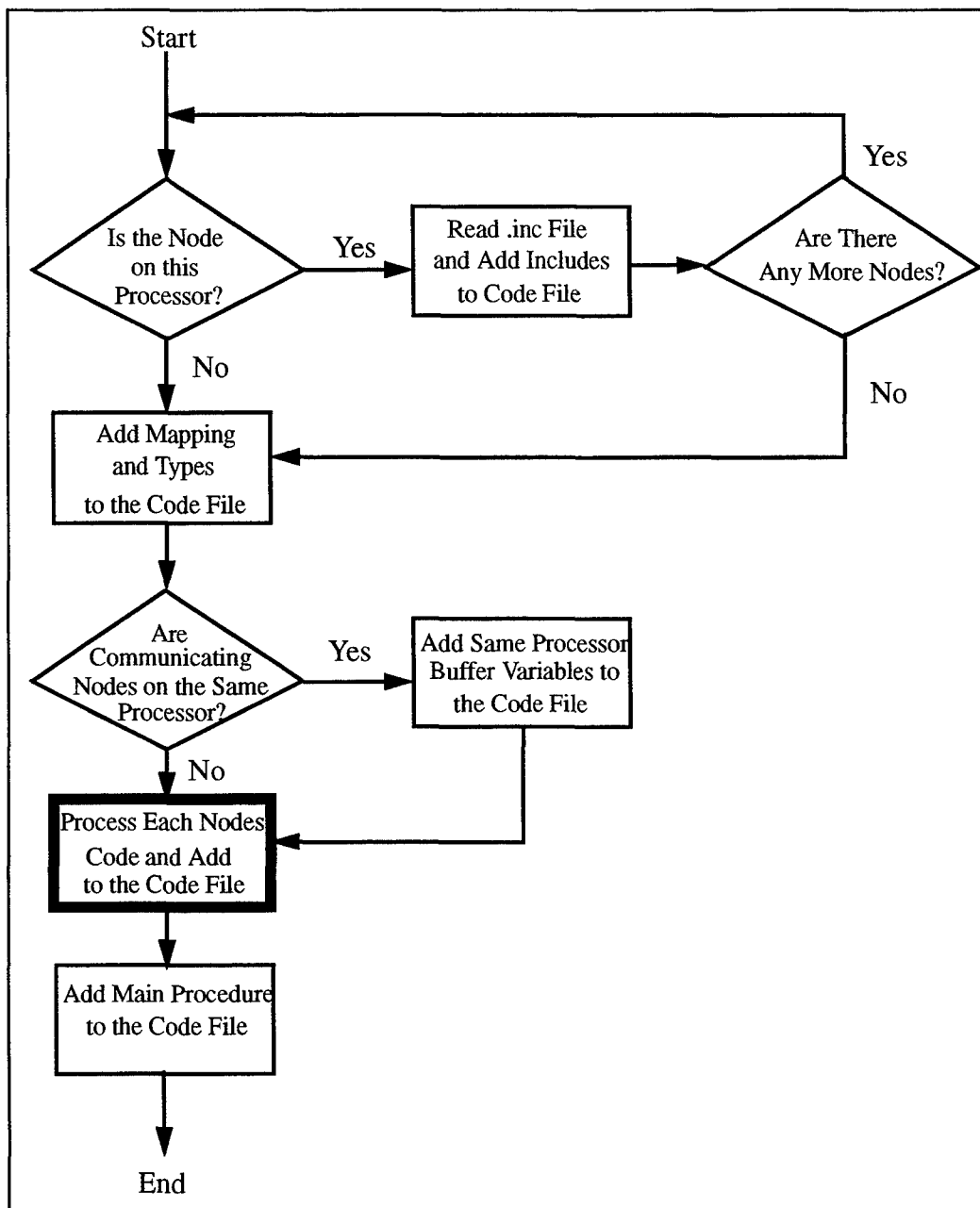


Figure 31: Execution flow for RPS code packaging. Step represented by the thick box is described in detail in Figure 32.

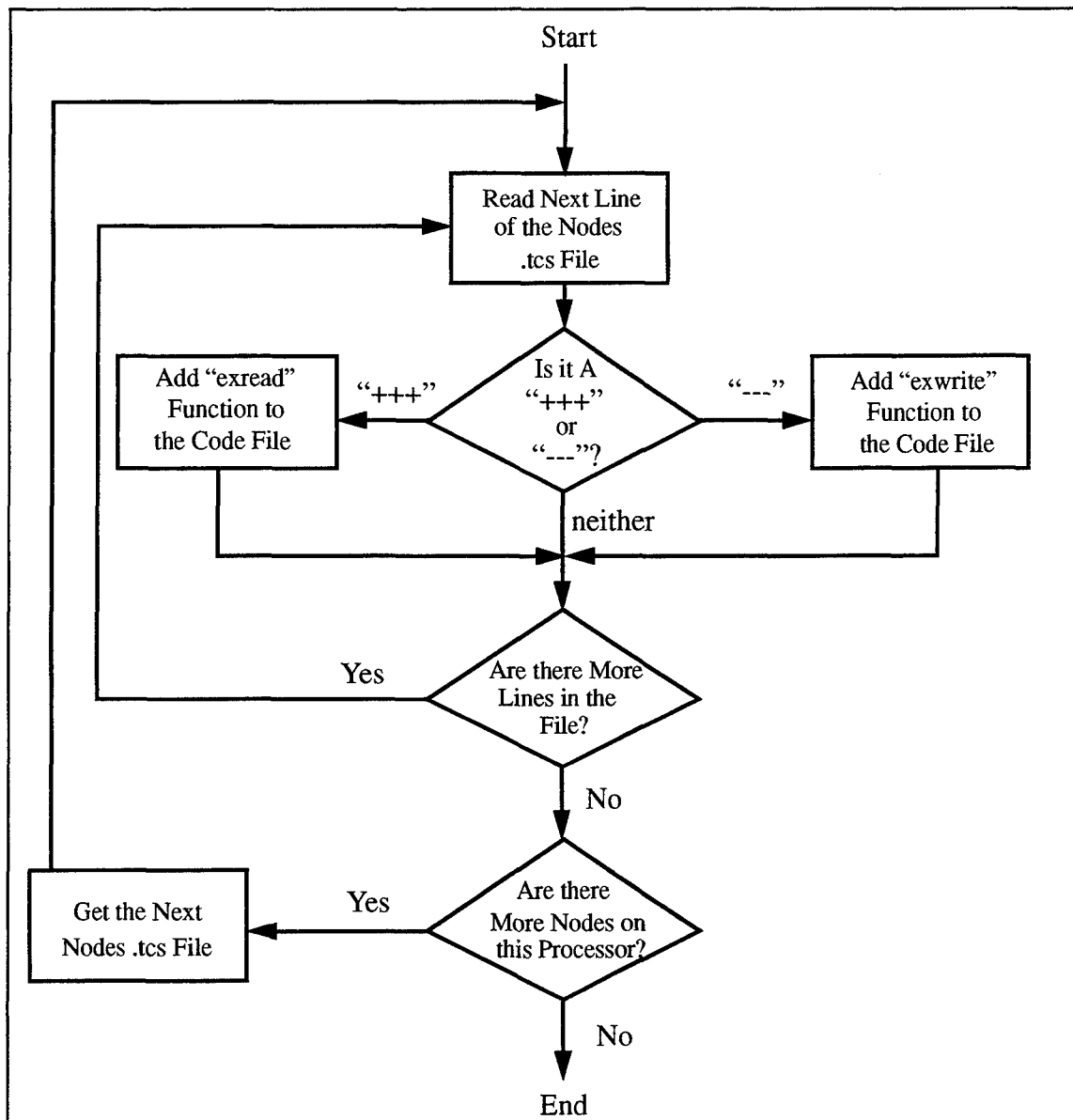


Figure 32: Flow of execution for processing node .tcs file. Describes step in Figure 31 represented by the thick box.

b. Node Requirements

The nodes may not use global variables. Structures that are used by the node must be defined in a header file which can then be included. If they are defined as a part the node file, same process buffering will not work causing compilation errors.

c. Node .tcs Files

The .tcs file must include several things as part of the code. These are a static integer variable index which is initialized to zero, an integer variable called loopcounter, and a statement which increments index at the end of the code. These lines are needed for same processor buffering code. If they are not included, errors will occur when the code is compiled or executed. The format for a nodes .tcs file is shown in Figure 33. This .tcs file is the file that goes with the .to and .inc files in Figure 29.

```
int
nodecode()
{
    static int index=0;    /* must be a variable declared exactly like this */
    int loopcounter;      /* must be a variable declared exactly like this */

    float val1;
    int val2[10];
    struct nodestruct val3; /* structure defined in node.h */

    /* the rest of the variable declarations */

    +++(&val1, sizeof(val1));

    /* the body of the node code */

    ---(&val1, sizeof(val1));
    ---(val2, sizeof(val2));
    ---(&val3, sizeof(val3));

    index++;              /* must appear as the last line before the return statement */

    return 0;
}
```

Figure 33: .tcs file format for node code.

d. Same Processor Buffering

If two nodes that communicate are scheduled on the same processor, a global variable is established for message passing and generated by the RPS packager. This eliminates much of the overhead associated with sending the message using message passing commands. This variable is a buffer that can be accessed by the two communicating nodes that are on the same processor. The buffer is established as an array that holds as many elements as needed to account for the different indices of the nodes. For example, if one node has an index of three and the other zero, four messages will be sent before any are read. Therefore, an array of four messages is established as a buffer.

The index variable described earlier is used by the node to keep track of the array element of the buffer that the message is to be read from or written into. By using a static variable, the counter is not reset when the node is completed. The loopcounter variable is used when the messages are array data types. This variable is used to copy the array elements into the buffer. These variables are not generated by the RPS packager and must be included in the .tcs file as shown in Figure 33.

This method is utilized in order to avoid large amounts of overhead associated with same processor message passing. As was shown in Chapter III, if a processor sends a message to itself, it requires in excess of 350 microseconds for the message to be sent and received. By using a designated buffer in memory for this, the overhead is reduced to the time of a memory access.

e. Node Indexing Effects

The node code that is packaged is done in such a manner that the nodes are executed according to the indices established by the revolving cylinder method. Because of these indices, the earliest iterations of the graph only execute nodes that have high indices and the latest iterations only execute nodes with low indices. This means that during some iterations of the cylinder, certain nodes do not get executed. If the nodes assigned to the processor were simply put in a loop, each node would be executed during every iteration

of the cylinder. In order to execute the cylinder properly, the iterations that do not require each node to execute are not included in the loop. The RPS packager generates code to execute these iterations of the cylinder before the loop begins and after it ends. The number of iterations that the loop makes is adjusted to reflect this. This is shown in Figure 34.

```

.
.
.
iterations -= 1;

timing[0] = extime();

node2();    /* node 2 has an index of 2 */

for (i=0; i<iterations; i++) {
    node2();
    node5();
}

node5();    /* node 5 has an index of 1 */

timing[1] = extime();
.
.
.

```

Figure 34: Iteration loop using indices.

4. Host PC Code

The host PC is responsible for loading the Transputer processes, starting their execution, releasing the Transputers once execution is complete, and all I/O functions of the program. In general, the format of all host programs will be similar. The main difference between host programs will be I/O. Because of this fact, most of the code generated is identical. This portion of the code is simply written to the file by the code generation program. The I/O portion of the program must be provided in a .tcs file. This file is interpreted by the RPS packager and the I/O function is generated from it.

The host PC also is responsible for getting the number of iterations of the task graph the user desires and broadcasting this to the nodes. An input statement followed by a

message broadcast statement which broadcasts the entered value to all processors in the system is placed in the code by the RPS packager.

a. Execution Flow

When creating the code for the host PC, the RPS packager first reads the .inc file for the host node and put appropriate #include statements in the host code file. Variables needed to gain access to the Transputers are then added to the code file. Global array variables are then added to the file which indicate the processor each node is mapped to and the type of each message in the same manner as the processor code files. After this, the .fil file is read and file pointers are added to the code so the I/O files can be accessed.

Each line of the nodes .tcs file is then read. After reading the line, it is examined to determine if it is a read or write. If it is a read, the schedule is examined to determine the index of the source node. A conditional statement is then added to the code file so that the read is executed during the proper iterations of the program. An "exread" command is added to the code file with the source processor and type of the message using the mapping and types variables. The next line of the code is then read and also placed in the conditional statement. This ensures that the output is produced only when the read is executed.

Likewise, if it is a write, the schedule is examined to determine the index of the source node. A conditional statement is then added to the code file so that the read is executed during the proper iterations of the program. The next line of the code is then read and placed in the conditional. This ensures that input is received prior to executing the message write. An "exwrite" command is added to the code file with the source processor and type of the message using the mapping and types variables.

The main procedure of the code is generated next. This includes local variables, and functions that control the number of iterations of the program, loading and running the Transputers, and setting the number of iterations for the program. The main procedure also include function calls to open and close any files used for I/O.

The flow of execution for creating host PC code by the RPS packager is shown in Figures 35 and 36.

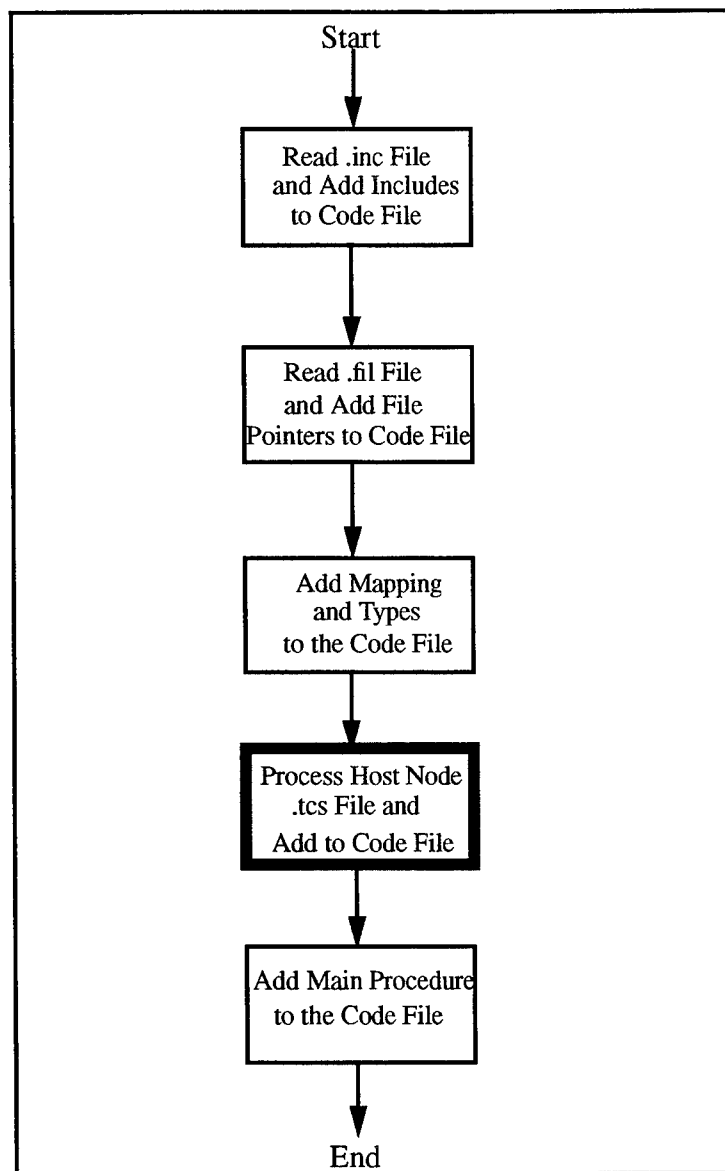


Figure 35: Flow of execution for host code packaging. Step represented by the thick box is described in detail in Figure 36.

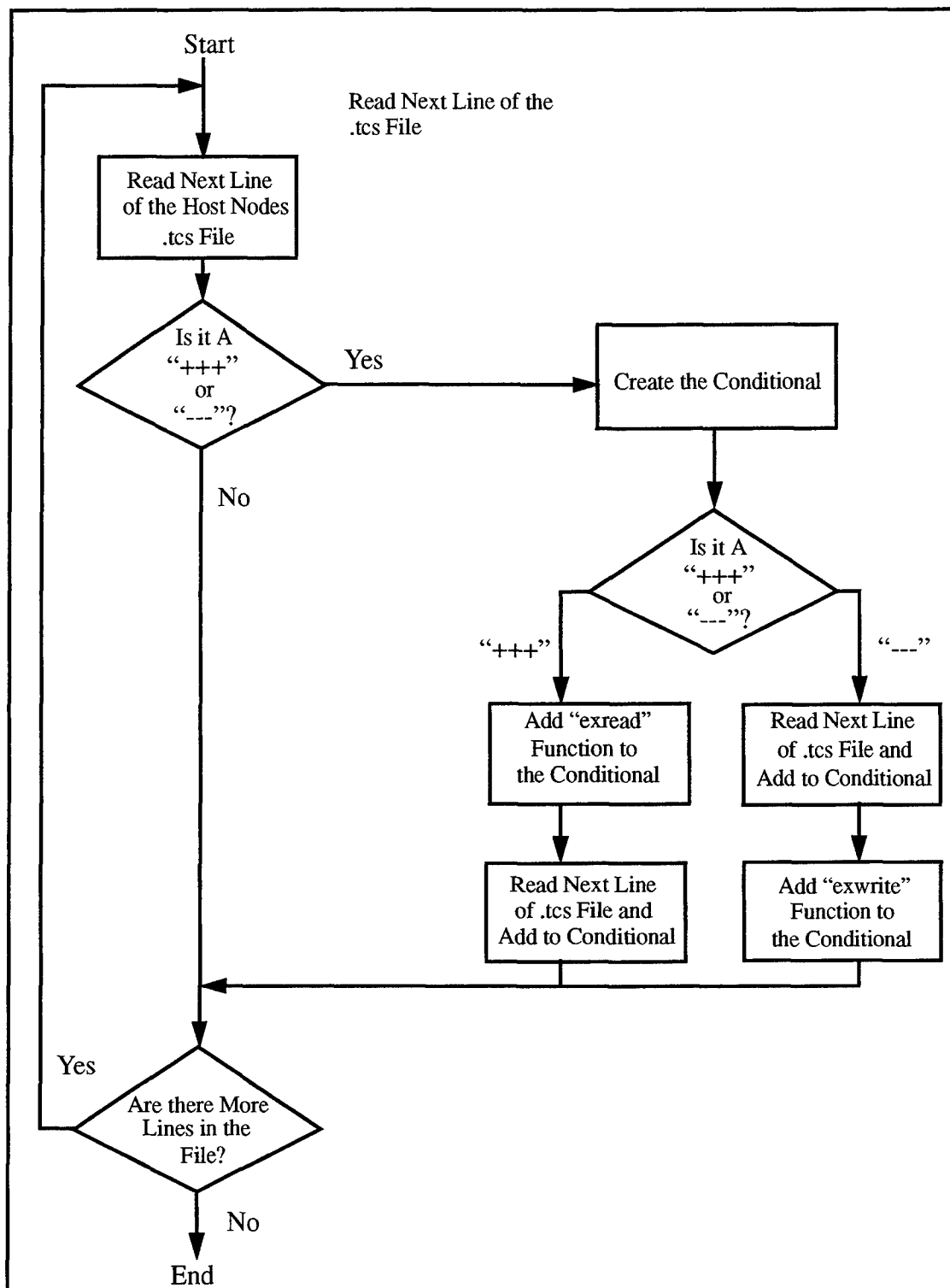


Figure 36: Flow of execution for processing host .tcs file. Describes step in Figure 35 represented by the thick box.

b. Message Passing and I/O for the Host

The host code file is generated in a similar way to the node code files. Each message that is sent or received by the host has a "---" or "+++" annotation associated with it. The RPS packager uses the graph and schedule files to determine the source or destination of each message and a unique type number is assigned.

The host node requires that the read or write annotation be written before the I/O command, either standard I/O or file access, regardless of whether it should be before or after. For example, if data is read from a file and then sent to a node, the logical order of statements would be to have the file read function first and the message send annotation second. The order of these must be reversed for the RPS packager.¹ The RPS packager will put it in the proper location, but the read or write command is needed by the packager to ensure that the proper data is retrieved from or sent to I/O. Also, if there is no I/O is to be performed, but a read or write is still executed, a blank line must follow the message passing annotation. The RPS packager expects an I/O command to follow message passing annotations and only a blank line will be interpreted properly. An example of a host .tcs file is given in Figure 37.

```
int
host()
{
    long val1, val2;

    ---(&val1, sizeof(val1));
    fscanf (infile1, "%d\n", &val1);

    +++(&val2, sizeof(val2));
    fprintf (outfile1, "%d\n", val2);

    return 0;
}
```

Figure 37: Example of host .tcs file.

1. This was discovered as an experimental fix.

c. Effect of Indices on I/O

When the code is generated for the host, it is done in such a way as so that the messages are sent or received in accordance with the indices assigned to the nodes sending or receiving the messages. Reads occur one iteration after the message is due to be received. This helps ensure that the host is not bogged down trying to receive a message which results in data that needs to be sent to the nodes being bogged down as well.

5. Timing and Synchronization

Timing and synchronization functions may be included in the code by selecting those options. Timing will cause the execution time of each processor to be displayed on the screen upon completion of the program. Synchronization will cause each processor to wait until all processors have completed the cylinder before continuing.

If a task graph is scheduled in such a way that it is not synchronized, the message passing buffers may overflow causing deadlock. This is alleviated by the using synchronization command. This also will result in extra overhead so it should only be used when necessary.

C. RPS PROFILER

The RPS profiler will generate code which, when compiled and executed, will determine the computation time of the tested node program. This information is necessary as it must be included in the graph file, and may be unknown to the programmer.

The RPS profiler generates code in a manner similar to the RPS packager and provides a batch file, compile.bat, for compiling the code. The RPS profiler generates a host program which loads and executes the generated node, and receives the execution time from the node. The node that is generated differs from the code generated by the RPS packager in the fact that all message passing annotations are removed. This way, the code only contains functions that are involved in computation.

When using the RPS profiler, the program will prompt the user for the name of the file to be tested. This file name is entered using no extensions. Since the node has three files, a

.tcs, a .to, and a .inc file, the profiler automatically adds the proper extension when needed. The user will also be prompted for the function name of the node. The complete name should be entered exactly as the function would be called. For example, if the function is called node it should be entered as node().

D. INPUT ERRORS

The RPS environment is still under development and currently is not tolerant of any errors in file format or programming. If there are any of these types of errors, the program will still run and produce output. However, the results are unpredictable and usually incorrect. If the output results in an unrealistic schedule, most likely one of the .cgd files is in the wrong format. If the schedule appears to be correct, but the code will not compile, an error in programming has most likely occurred.

V. TESTING AND RESULTS

A. METHODOLOGY

Experiments were conducted to determine the effectiveness of RPS at maximizing the throughput of repetitive task graphs. The results of these experiments were compared with the predicted results and with the MH heuristic developed by El-Rewini and Lewis [ELR 90]. Since MH is designed to minimize response time and RPS minimizes the average execution time per instance, the comparison is performed by repeating application of the schedule determined by MH to successive instances of the task graph, even though MH does not consider pipelining of successive instances of the graph. A comparison between the two different RPS variations is also made.

1. Benchmarks

The experiments were conducted on both real and randomly generated task graphs. The graphs were used to write programs that simulated the execution of the task graphs. The programs were then profiled, scheduled and packaged using the RPS profiler, scheduler and packager.¹ The code was compiled and run and execution time was recorded.

Two actual algorithms were used in the experiments. One, the correlator graph, is a signal processing application, and the other, Gaussian elimination, is a popular linear algebra algorithm. By using these algorithms, the effectiveness of RPS on actual applications can be shown.

The six random graphs were generated as layered task graphs such that each node of a layer is connected to at least one node in the preceding and succeeding layer. Of these graphs, three contain 20 nodes and approximately 30 edges, and the other three contain 40 nodes and approximately 55 edges. Each graph consists of a different number of levels. This ensures that the spectrum of graphs, from highly parallel to highly sequential, that most likely resemble actual task graphs are tested.

1. Since the programs were written to simulate the task graphs, the nodes had to be profiled to ensure that they accurately modeled the nodes of the graph.

Each of the graphs were modified to vary the computation-communication ratio by varying the size of the messages. The graphs were changed so that the ratio was changed, but the graph maintained the same topology. The computation-communication ratios used were 10 to 1, 5 to 1, and 1 to 1. The amount of communications was estimated by using only the time taken to transfer a byte, not the overhead associated with it.

The 20 node graphs were also modified such that the computation-communication ratio remained constant, but the magnitude of both computations and communications were increased. By doing this, the effect of multiple block messages was examined.

2. Multiprocessor Topology

Two different system topologies were used in the experiments, the ring and the hypercube. The ring topology is an interconnection network that is most restrictive of communications. The hypercube represents a moderately connected network. Each of these configurations were tested using two, four, and eight processors.

3. Scheduling Methods

The RPS scheduler provides the two different scheduling variations described in Chapter IV. For each of the benchmarks tested, both scheduling methods were utilized.

B. EXPERIMENTS

The measure that is used for comparisons in the experiments is efficiency. Efficiency is defined to be the speedup found divided by the number of processors. This way, a comparison between speedups found using a different number of processors can be made.

1. Accuracy of the System Model

The test programs were run on the Transputer system and the execution times were recorded. In order to determine how accurate the system model that was used is, the actual execution times of the graphs were compared to the execution time that was predicted by the RPS scheduler.

The difference between predicted and actual execution times varied mostly according to the computation-communication ratio of the graph. A comparison between the three different ratios showed that a 10 to 1 ratio produced actual execution times that varied from predicted in the range of 5 percent of the predicted time. For a 5 to 1 ratio, the variation increased to 10 percent. For a 1 to 1 ratio, the variation increased further to 25 percent.

The communications model that was achieved through testing generated equations that were used to determine message passing times and routing delays. Calculations made using these equations yield results which tend to be accurate to within 10 to 20 microseconds. These differences would account for actual execution times which were not exactly the same as predicted, but would not account for the large variations found.

Since the largest variations are encountered when the communications of a task graph are increased, the most likely cause of the variation is that our communications model is incomplete. Since an actual system model was unavailable, tests were performed to identify the model. While these tests gave a good indication of the underlying model of the system, it is likely that one or several factors which affect communications are not examined by our testing method.

2. Efficiency of the Scheduling Heuristic

The efficiency of the RPS heuristic can be judged by examining how effective it is at increasing the throughput of the task graph. In these experiments, we look at the efficiency of the simple heuristic. The measure we will use for this is efficiency. The efficiency is the speedup attained divided by the number of processors that are used.

The effectiveness of the algorithm is relative. While the algorithm may produce increased throughput, its effectiveness can only be judged by comparing it to another algorithm. This is done in a later section. In this section, we compare different graph characteristics to show what type of graphs RPS most effectively schedules. For this set of experiments, the simple heuristic is used.

Since computation-communication ratio is often discussed in this chapter, the quantity R will be used in the figures to denote computation-communication ratio. All values described by R are a ratio of time values. For example, $R=10\text{to}1$ describes a computation-communication ratio where the graph averages 10 time units of computation for every 1 time unit of communication.

a. Random Task Graphs

For our set of random task graphs, we divide them into two groups. Group 1 contains the graphs with 20 nodes and group 2 the graphs with 40 nodes. The average computation time per node for each task graph is set to 2,000 microseconds. The efficiency of each graph was measured on different system topologies utilizing two, four, and eight processors.

Figure 38 shows the efficiencies of the two groups on a hypercube topology when each graph utilized a 10 to 1 computation-communication ratio. The average communications is set at 200 microseconds to achieve this ratio. As we can see from the graph, the graph containing more nodes have a higher efficiency than graphs with fewer nodes. This is as expected since it is more likely that the nodes will be able to be evenly distributed amongst the processors when more nodes are present.

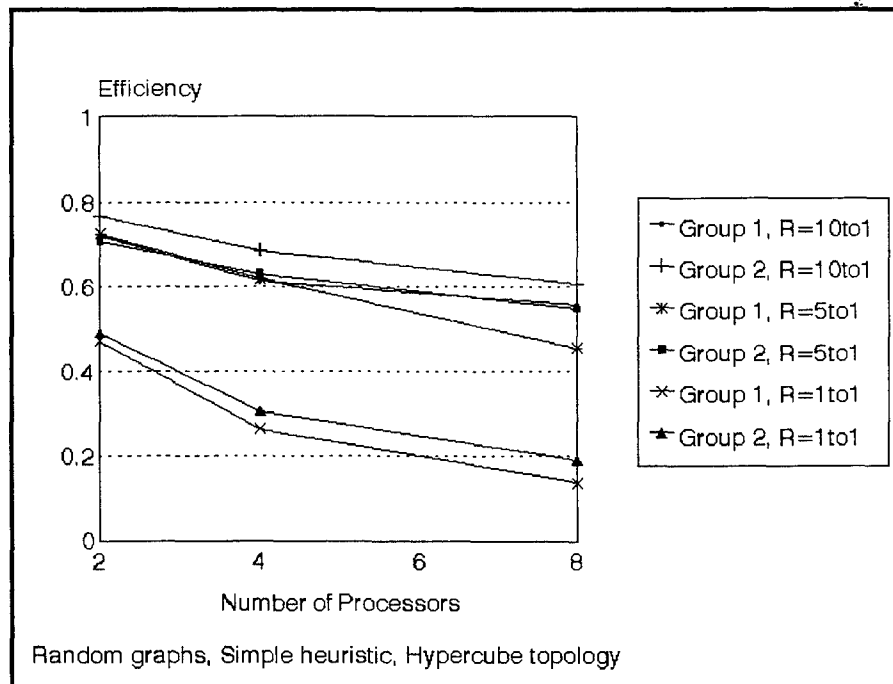


Figure 38: Efficiency for random task graphs using hypercube topology and simple heuristic.

Figure 38 also shows the same relationship between the two groups of graphs where the computation-communication ratio was altered to 5 to 1 and 1 to 1 by increasing the average communication to 400 and 2,000 microseconds, respectively. The same trend that was seen with the 10 to 1 ratio is again shown, more graph nodes leads to higher efficiency.

Looking at the speedup found for the graphs in group 1 for the different computation-communication ratios we see that increasing the communication decreases the amount of speedup that the algorithm is able to produce. Since the communications requires processor time, this processor time is no longer available to computation, thus

reducing the maximum throughput that can be achieved. By the time the ratio reaches 1 to 1, the algorithm provides little if any speedup. This is depicted in Figure 39.

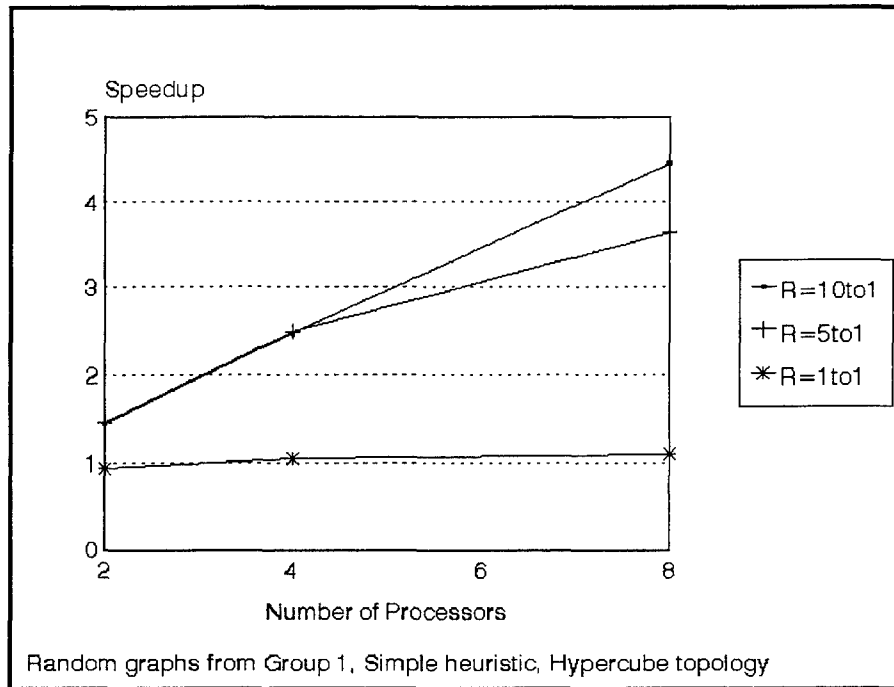


Figure 39: Speedup for random task graphs from group 1 using hypercube topology and simple heuristic.

Since the message passing time is non-linear, a test was conducted to determine if increasing the scale of the task graph (increasing both computation and communication while keeping the computation-communication ratio the same) had any effect on the efficiency. Using the group 1 graphs and a 10 to 1 computation-communication ratio, the average computation time was increased to 20,000 microseconds and the average communication time was increased to 2,000 microseconds, thus keeping the 10 to 1 ratio. These graphs are referred to as large scale while the original graphs are referred to as small scale. Figure 40 shows the resulting efficiencies when run on a hypercube topology.

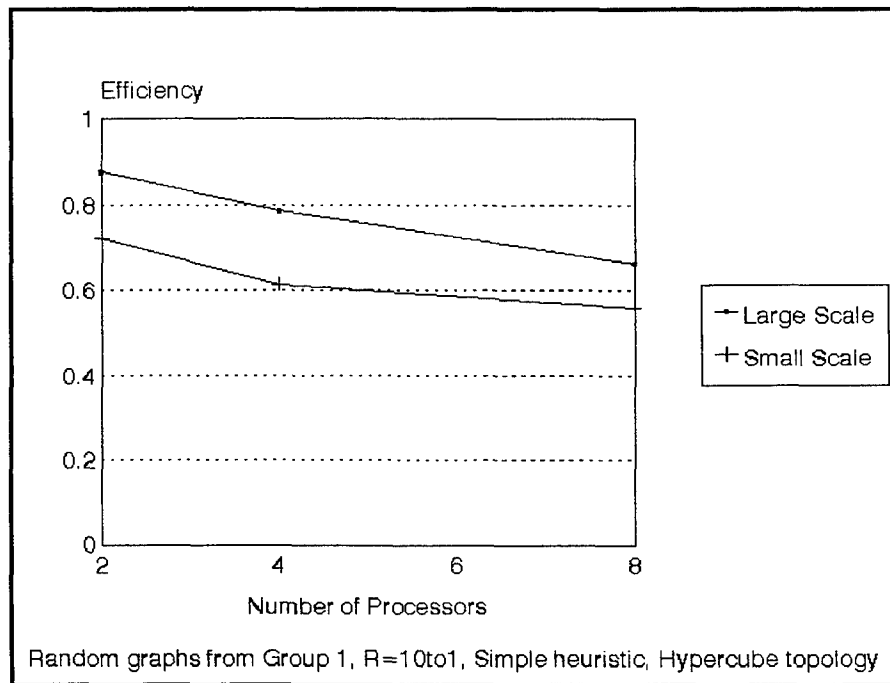


Figure 40: Efficiency of large scale vs. small scale random graphs from group 1.

As can be seen, graphs which utilize larger computation and communication times achieve greater efficiency when scheduled using RPS. This can be attributed to message pipelining that is seen when messages consist of more than one block. Also, the message overhead is not as much of a factor since it becomes small in comparison to the computation time.

Finally, a comparison between the hypercube and ring topologies is made for both groups of graphs. The efficiencies are depicted in Figure 41.

As with the hypercube, the ring configuration also has the characteristic that a graph with more nodes executes with a higher efficiency. By comparing the ring and hypercube, very little difference is seen, and in some cases, the ring performs slightly better than the hypercube. Since the ring has a more restricted communications network, this is not expected. The most likely cause of this lies with our algorithm. Because we use a greedy algorithm, once a task is scheduled, its mapping cannot change. This affects the mapping

of tasks that are scheduled later and may cause them to be placed on processors that are less optimal on a hypercube.

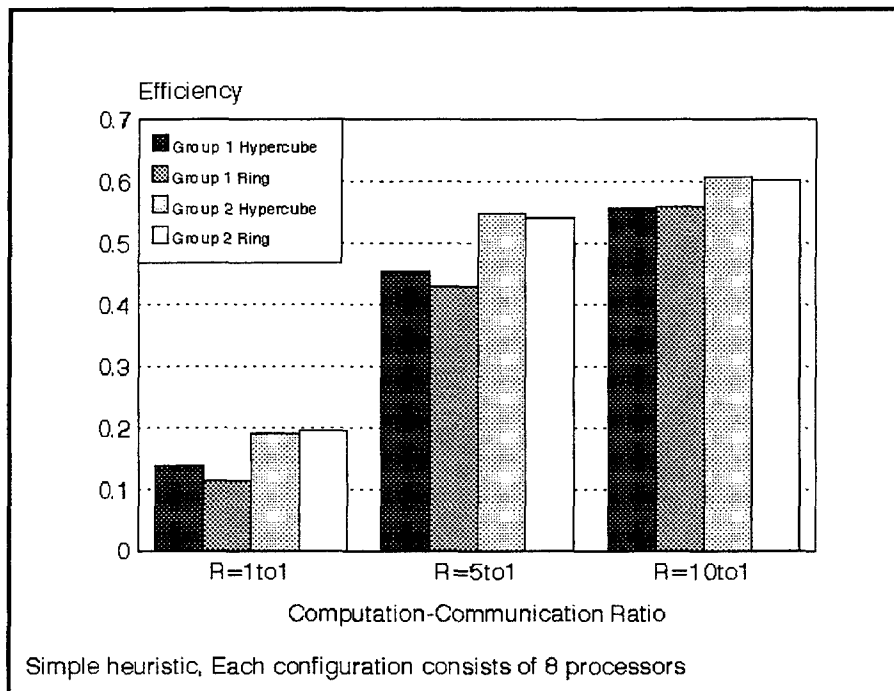


Figure 41: Comparison of 8 processor ring and hypercube topology for random task graphs.

b. Correlator Graph

The correlator graph was tested to provide evidence that the RPS heuristic is effective at scheduling real signal processing applications. The tests performed on the random task graphs were used for testing the correlator. Neither the computation-communication ratio nor the scale of the scale of the task graph were altered since the correlator application is a real application and its ratio is fixed. The ratio of the correlator is 6.21 to 1. The correlator task graph is depicted in Figure 42. The computation numbers for each node are in microseconds and the communication numbers are in bytes.

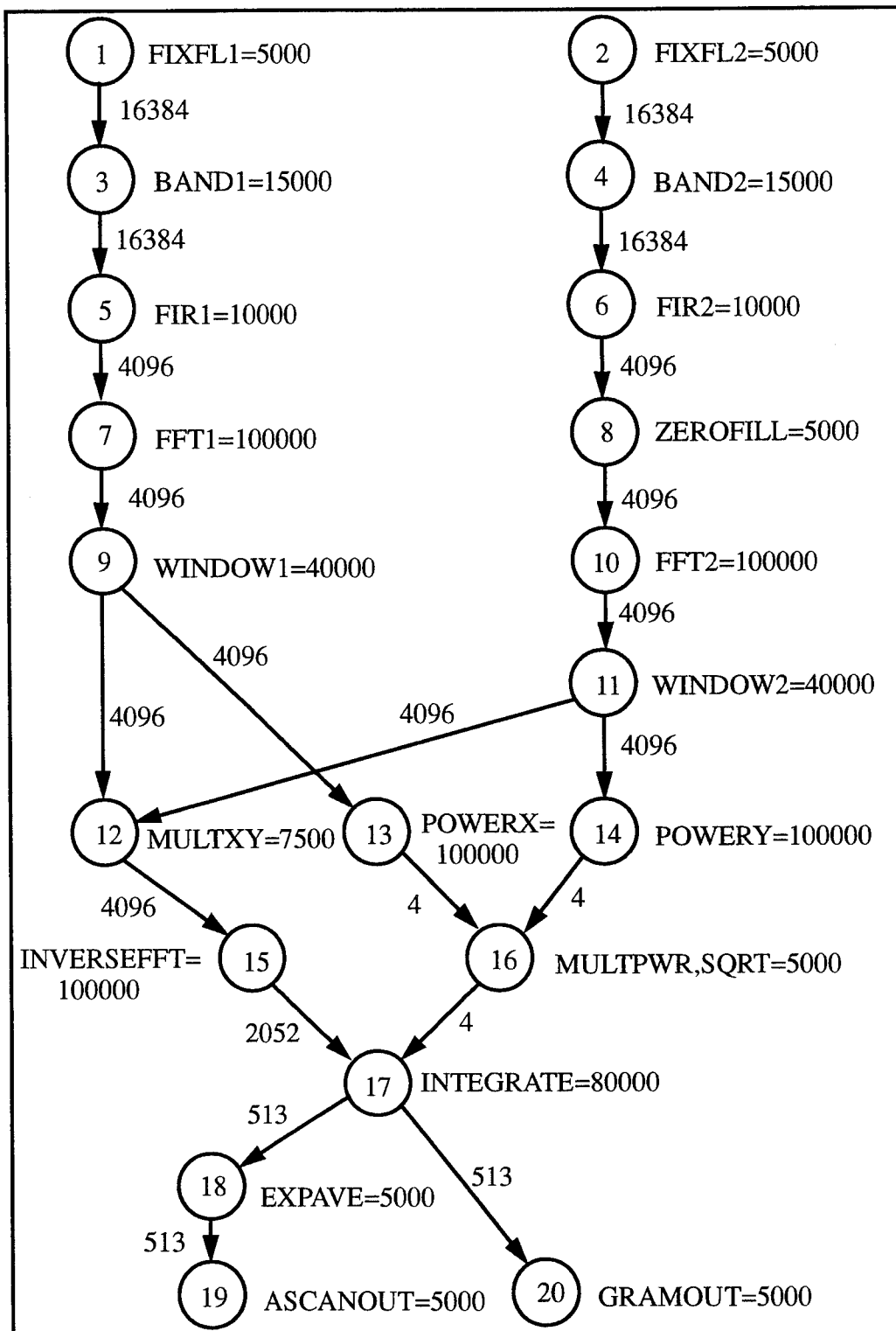


Figure 42: Correlator task graph. From [SHU 92].

The resulting efficiency of the correlator is shown in Figure 43 along with the efficiency of the random graphs with 10 to 1 computation-communication ratios. The correlator performance follows the trend shown with the random graphs and produces even better results than the random graphs. This is most likely due to the structure of the correlator graph. With minimal dependencies between nodes, the scheduler has more flexibility with processor assignment.

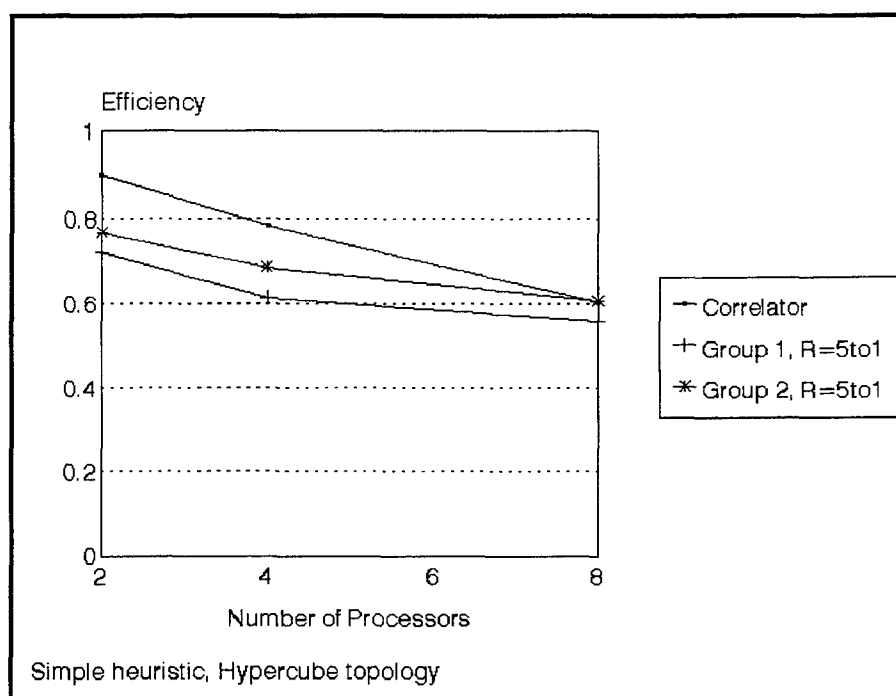


Figure 43: Efficiency of correlator vs. random task graphs with R=5to1.

The hypercube and ring topologies are then compared for the correlator. This is shown in Figure 44. The correlator shows a more dramatic decrease in efficiency when using a ring topology than was seen with the random task graphs. This is most likely caused by the correlator being mapped in such a way that longer routes were used for message passing.

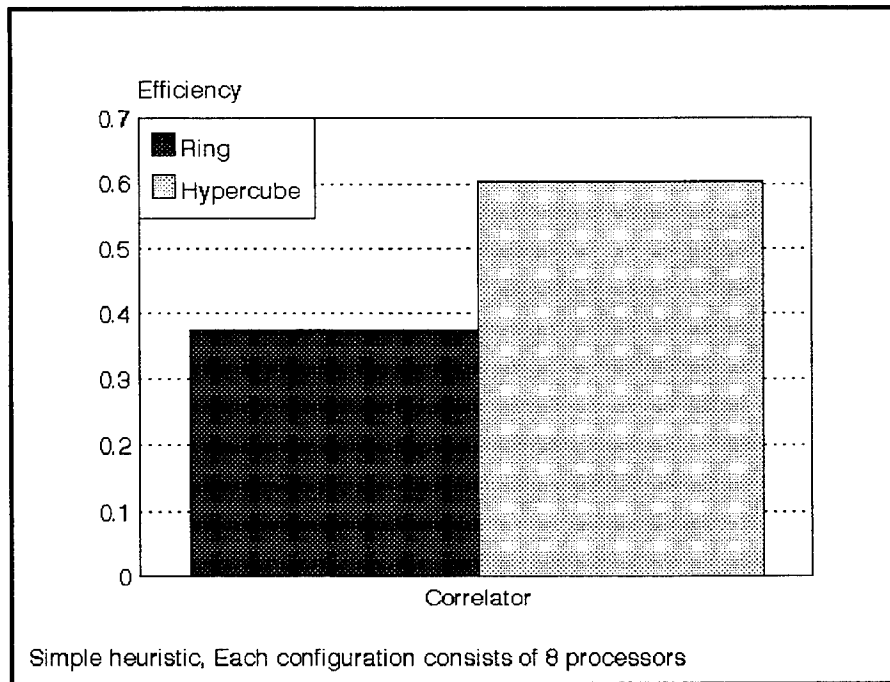


Figure 44: Comparison of 8 processor ring and hypercube topology for correlator graph.

c. Gaussian Elimination

The Gaussian elimination graph, a linear algebra application, was tested to provide further evidence that RPS is effective at mapping real applications. The Gaussian elimination task graph is shown in Figure 45.

When testing Gaussian elimination using the actual computation and communication values given, the resulting mapping had all nodes assigned to the same processor. This, of course results in no speedup. Because the message passing overhead is much larger than the computation time of the node, placing all nodes on the same processor and avoiding the overhead provided the best schedule.

To examine whether communication overhead is the only reason that no speedup is achieved with Gaussian elimination, the scale of the program was increased. Each node size and message size were increased by 10 times and then by 50 times. While this resulted in a mapping that place some nodes on different processors, the resulting speedup was

minimal. Examining the computation-communication ratio, we can see the problem. The ratio is 1 to 1.54 which is even worse than 1 to 1. As was shown with the random graphs, little to no speedup is achieved with a 1 to 1 ratio.

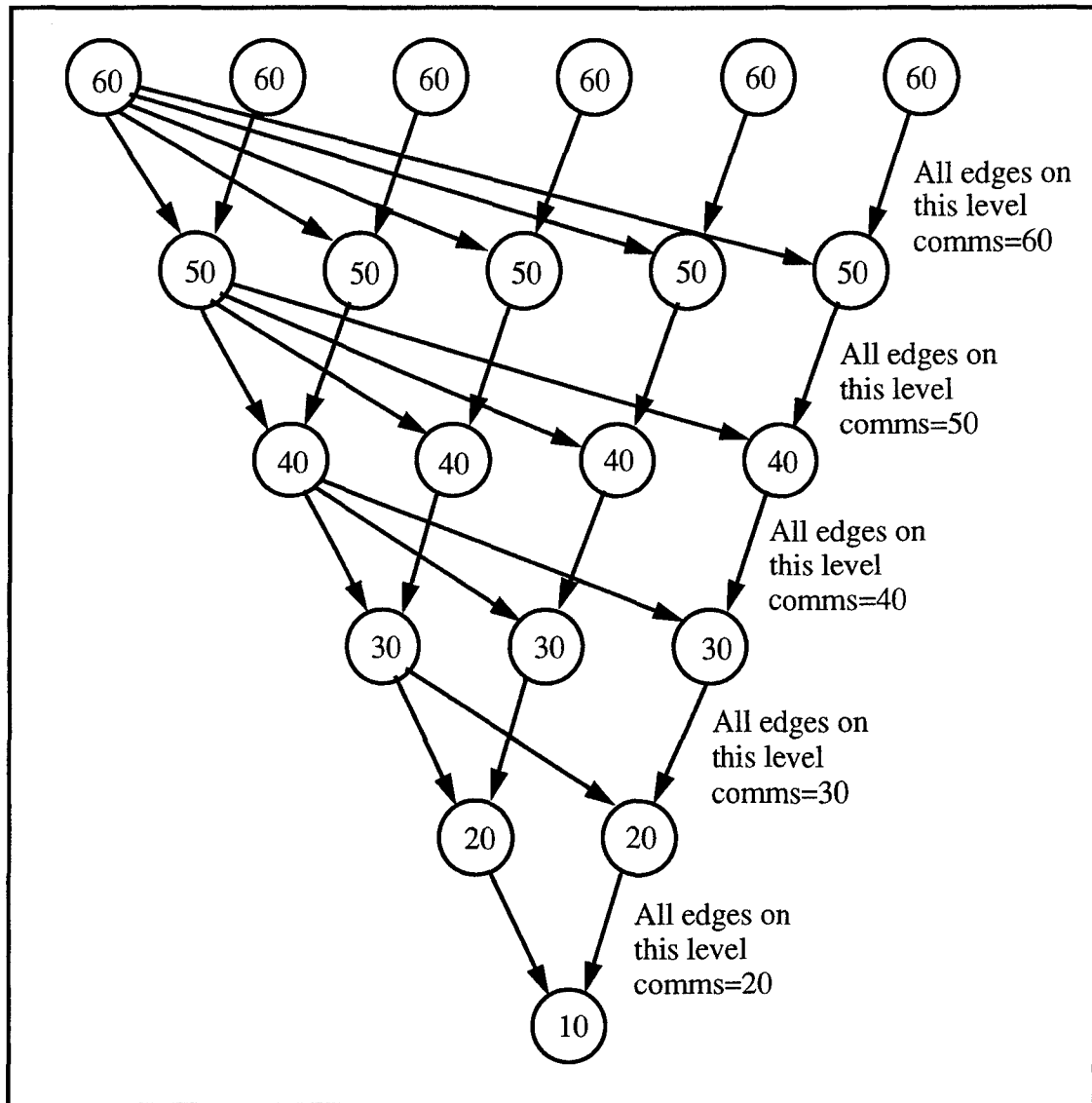


Figure 45: Gaussian elimination task graph. From [DIX2 93].

Since the amount of communications is causing no speedup to be gained by RPS, another possible solution arose. If the speed of the communication links could be increased, speedup comparable to the other graphs might be achieved. To simulate this, the two larger

scale graphs were alter such that each was tested with a 10 to 1 and a 5 to 1 computation-communication ratio. This simulated faster links. The resulting efficiencies are shown in Figure 46.

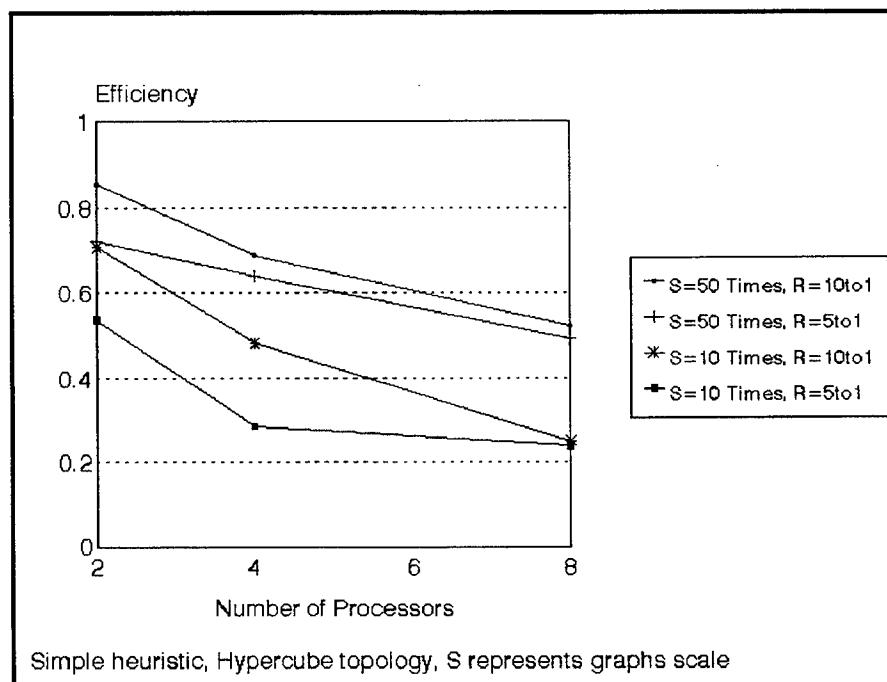


Figure 46: Efficiency for Gaussian elimination task graphs.

The resulting efficiencies again closely follow the random task graphs. Increases in the scale of the graph and increases the computation-communication ratio result in increased efficiencies.

The ring topology is compared with the hypercube for these graphs. This is shown in Figure 47. For Gaussian elimination, only small differences are found between the ring and hypercube. As was seen with the random graphs, in some cases RPS provides better mapping for the ring than the hypercube.

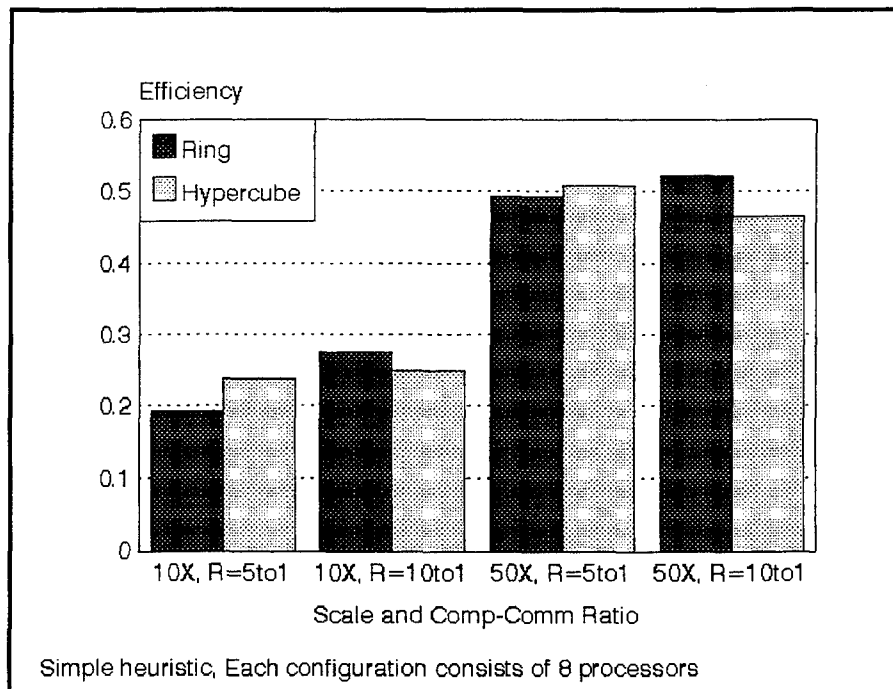


Figure 47: Comparison of 8 processor ring and hypercube topology for Gaussian elimination graph.

3. Comparison of the Two Scheduling Variations

The two scheduling variations tested with RPS are of different complexity for mapping the tasks to processors. The simple heuristic uses a simple method of placing a task on the first processor with the lowest maximum resource utilization, while the complex heuristic breaks ties by using the other resource utilization figures. The complex heuristic requires that the resource utilization figures be sorted so that if a tie exists, the second highest utilization, and lower utilization if necessary, of these processors can be compared.

Each iteration of method one consists of assigning the task to a processor, update the appropriate utilization tables, and repeat for each processor in the system. This is repeated for each task. If we have P processors, updating the utilization tables is proportional to P . This means that if we have T tasks, the complexity of the simple heuristic is $O(TP^2)$.

For the complex heuristic, after the utilization tables are updated, the figures must be sorted. Since the size of the tables is proportional to P and the most efficient sort is

$O(P \log P)$, sorting the utilization figures is of order $O(P \log P)$. This means that the complex heuristic is of order $O(TP^2 \log P)$.

Comparing the results of the two methods shows that the complex heuristic cannot guarantee a better schedule than method one. For example, one of the 20 node graphs had a speedup of 3.95 using the simple heuristic and 3.86 using the complex heuristic for an 8 processor hypercube. This same graph had a speedup of 3.74 using the simple heuristic and 3.88 using the complex heuristic for an 8 processor ring.

Because of the additional complexity of the complex heuristic, the simple heuristic is the better method to use. For a large number of processors, the running time of the complex heuristic would greatly exceed the simple heuristic with no guarantee of improved performance.

The only case where the complex heuristic consistently gave better results than the simple heuristic is for a 1 to 1 computation-communication ratio. Since we have seen that this ratio does not give significant speedup anyway, graphs with this ratio would most likely not be used with RPS.

4. Comparison with MH heuristic

Comparing RPS to MH was done by utilizing the MH simulator that was constructed by Kasinger on our task graphs. This simulator was constructed using a simplistic model for the underlying system hardware and software and does not take into account all factors of message passing that are present in our system. In order to compensate for this, the task graphs were adjusted to reflect these message passing factors. While this simulation gives a good comparison between the two methods, actual testing on the system would provide more accurate results.

a. Random Task Graphs

The two groups of random task graph were run on the MH simulator using the hypercube topology. The resulting efficiencies are shown in Figure 48 for graphs with a 10 to 1 computation-communication ratio. Also depicted are the efficiencies found using RPS.

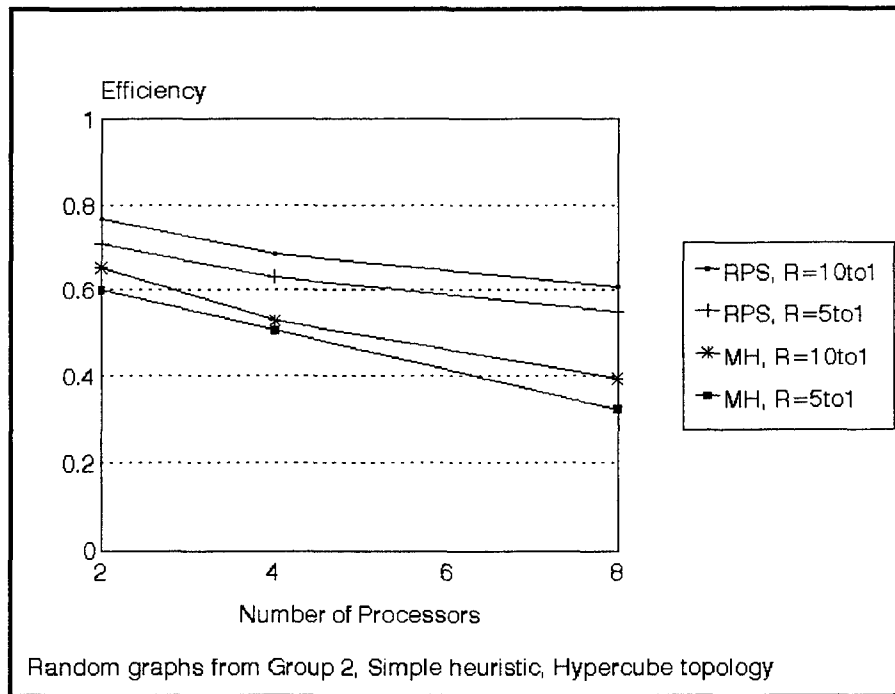


Figure 48: RPS vs. MH using the hypercube topology for random task graphs.

RPS provides a better mapping than MH for any number of processors. This improvement tends to increase as the number of processors increases. Of all graphs tested, in only two cases did MH provide a mapping that resulted in a better mapping than RPS. These cases were for 2 processors and the resulting efficiencies were nearly identical.

Tests were also run using a 5 to 1 computation-communication ratio, shown in Figure 48. These results were similar to the 10 to 1 ratio. A 1 to 1 ratio was not test since little speedup was achieved with RPS anyway.

Another interesting note is that the graphs that had a structure that was had more parallelization performed better on MH than ones that are more sequential. This is due to the fact that MH does not consider pipelining when determining the mapping. Even with this improved performance, these graphs still did not perform better than RPS.

The ring configuration was also tested with similar results. Figure 49 shows the 10 to 1 ratio comparison of RPS and MH. The 5 to 1 ratio, also shown in Figure 49, also resulted in RPS providing more efficiency than MH.

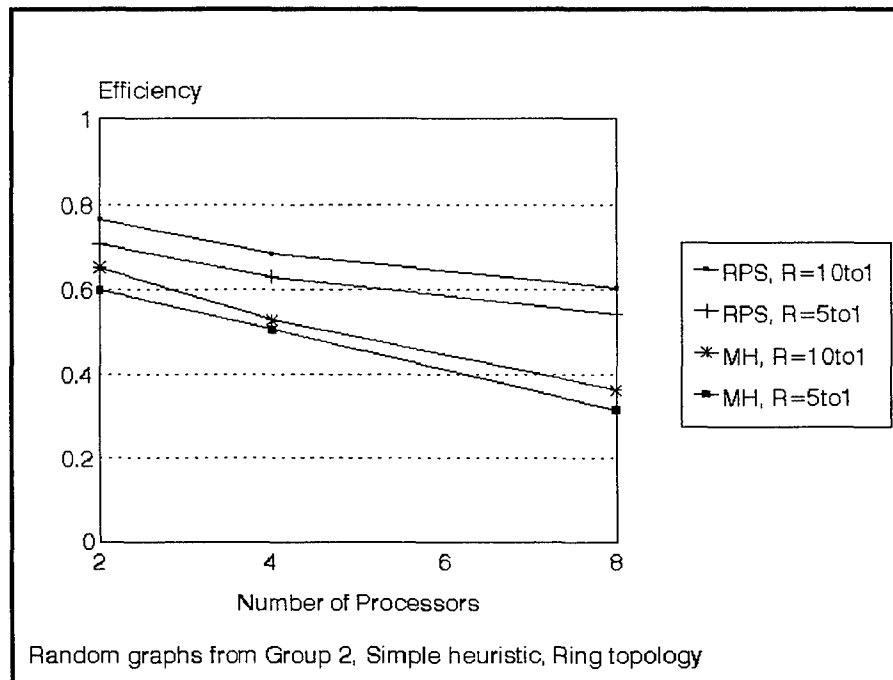


Figure 49: RPS vs. MH using the ring topology for random task graphs.

b. Correlator Graph

Similar comparisons were made for the correlator graph. In all cases for the correlator graph, RPS provided a greater efficiency than MH. This held for both the hypercube and the ring topologies. The comparison of RPS and MH for the correlator on the hypercube is shown in Figure 50.

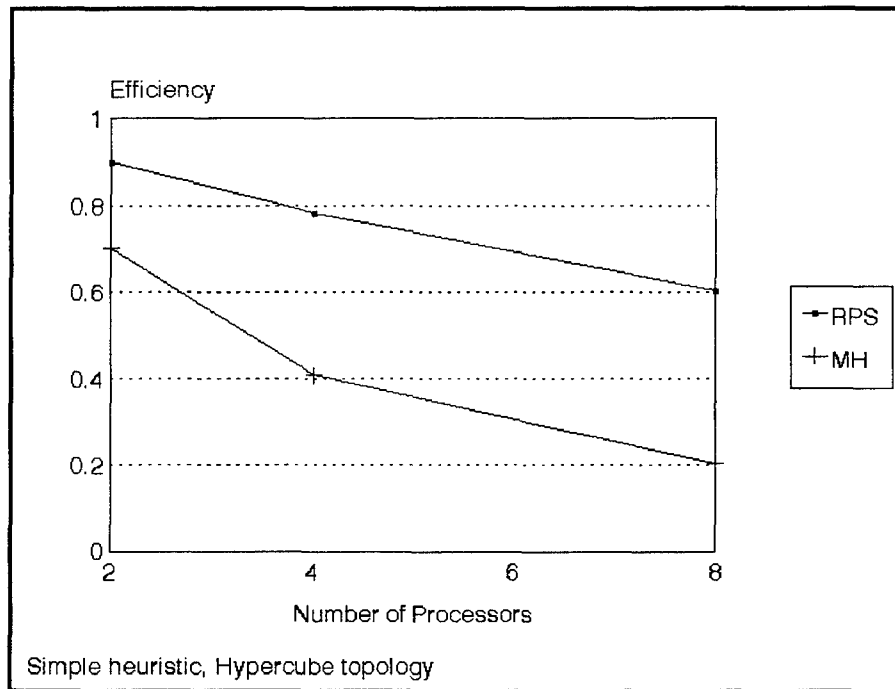


Figure 50: RPS vs. MH for correlator graph on hypercube topology.

c. Gaussian Elimination

The comparison between RPS and MH for Gaussian elimination were made using the two larger scale graphs with 10 to 1 and 5 to 1 computation-communication ratios. The results for the hypercube topology are depicted in Figure 51. These results are similar to the results found for both the random graphs and the correlator. The ring configuration also provided similar results. A 1 to 1 ratio was not tested since this ratio, again, showed no significant speedup using RPS.

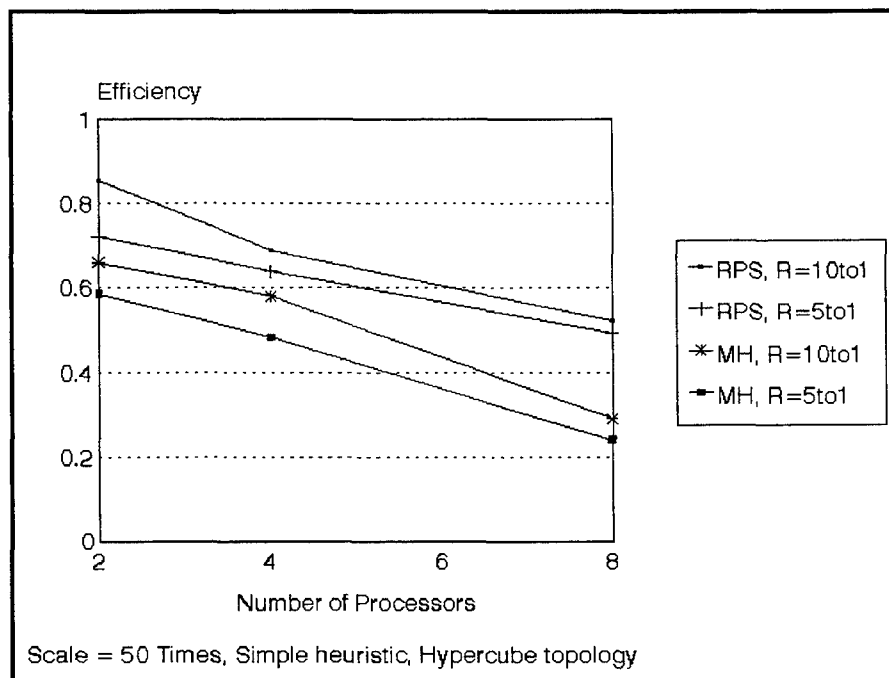


Figure 51: RPS vs. MH for Gaussian elimination on hypercube topology.

VI. CONCLUSIONS

A. CONCLUSIONS

This thesis presented the Realistic Periodic Scheduling (RPS) heuristic and the programming tools and environment that implement the heuristic. The RPS heuristic is designed to maximize the throughput of repetitive task graphs on a distributed memory multiprocessor, as opposed to minimizing the execution time of one iteration. The heuristic takes into consideration system topology, communications between tasks, and resource contention in determining a schedule as in Kasingers [KAS 94] PS heuristic. Also considered are the characteristics of the underlying system hardware and software that affect communications and execution time. By targeting repetitive task graphs with the RPS heuristic, pipelining of successive iterations of the graph is possible. RPS takes advantage of this.

The RPS heuristic is described in detail in this thesis. Also described are the programming tools which implement the heuristic. These are the RPS scheduler, RPS packager, and RPS profiler. These tools were used to create, profile, schedule, and package repetitive task graphs. Tests of these graphs show that RPS is an effective method of scheduling repetitive task graphs. Average efficiencies of 67 percent on four processors and 59 percent on eight processors using a computation-communication ratio of 10 to 1 were observed, while lowering the computation-communication ratio resulted in lower efficiencies. The communications model used proved to be adequate as the actual execution times of the graphs were close to the predicted times. These times were within 5 percent for 10 to 1 computation-communication ratios and 25 percent for 1 to 1 ratios. Comparisons between the simple and complex heuristic show that no appreciable throughput improvement is gained by the more complex algorithm. The average difference was less than 5 percent. Comparisons with the MH heuristic of El-Rewini and Lewis [ELR 90] show that superior throughput can be achieved by RPS. Efficiencies observed using RPS were an average of .14 higher than MH on four processors and .21 higher using eight processors. A

case where RPS is particularly effective is larger scale graphs. RPS is particularly advantageous over MH for larger numbers of processors and graphs that are highly sequential because of its overlapping of different graph instances. Using a highly sequential random graph and eight processors, the efficiency observed using RPS was .31 higher than MH.

B. FUTURE WORK

Additional research on RPS is needed for task graphs containing cycles. The model that was used by RPS constrained the graphs such that the graphs must be acyclic. Enabling task graphs with cycles to be scheduled by RPS opens a wider base for the use of RPS.

Research should also be conducted in the area of incorporating granularity management techniques [NEG 94] into the heuristic. As was seen in the tests, changing the node sizes has an effect on the resulting schedule and throughput achieved. By incorporating granularity management into the heuristic, higher throughputs would be possible.

A Graphical User Interface (GUI) should be incorporated with the system. This would aid the programmer in developing task graphs for use with RPS.

Finally, the RPS heuristic should be implemented on other systems with different hardware and software characteristics. This would help show the validity of RPS, regardless of the system used.

APPENDIX

This appendix contains the C code for all scheduling, code generation, and computation time testing functions found in this thesis.

A. RPS SCHEDULER AND RPS PACKAGER MAIN PROGRAM

```
/* This is the main program for RPS Scheduler and RPS
   Packager */

#include "codegen.h"
#include "schedule.h"

int
main()
{
    generateSchedule();

    codeGen();

    return 0;
}
```

B. SCHEDULE FUNCTIONS HEADER FILE

```
/* This file is schedule.h. It is the header file for the RPS scheduling
   functions */

#ifndef __SCHEDULE_H
#define __SCHEDULE_H

#define MAXPROCS          16
#define MEM_ACCESS        3
#define BYTE_MEM_ACCESS   .1274
#define INST_OVHD         150
#define SETUP             155
#define ROUTING           70
#define BLOCK_SIZE        1024
#define BYTE1             1.23
#define BYTE2             1.1
#define BYTE3             0.98
#define RTNG_DLY          54
#define BYTE_RT_DLY       .098

struct schedData {    /* structure used to hold schedule data that is */
    int    processor; /* generated by the scheduling program          */
    int    index;
};
```

```

    int    order;
    long   startTime;
    long   finishTime;
    int    overheadTime;
    int    commTime;
    int    compTime;
};

/* Schedule generation function. It uses all other functions in this
   file to generate the schedule */

void generateSchedule ();

/* Reads the graph file and stores computation times in times,
   communications times in edgemat, and the number of nodes in the
   graph in nodes. */

void readGraph (int *edgemat, struct schedData *sched, int nodes);

/* Reads the routing configuration file and stores the number of
   processors in procs and the routing between any two processors in
   the routes matrix. */

void readRouting (int *routes[MAXPROCS+1][MAXPROCS+1], int procs);

/* Outputs the schedule data to the output file in a form that is
   readable by the code generation functions */

void outputSchedule (struct schedData *sched, int nodes, int procs,
                    int *rDelay);

/* Schedules the nodes to processors using the Periodic Scheduling
   Heuristic*/

void PSHeuristic (int *routes[MAXPROCS+1][MAXPROCS+1], int *edgemat,
                 struct schedData *sched, int procs, int nodes,
                 int *routeDelay);

/* Determines the order that nodes are to be scheduled according to
   largest available node first */

void determineOrder (int *ord, int *edgemat, struct schedData *sched,
                   int nodes);

/* Copies one set of utilization tables to the other */

void copyTables (long linksrc[MAXPROCS+1][MAXPROCS+1],
                long procsrsrc[MAXPROCS+1],
                long linkrec[MAXPROCS+1][MAXPROCS+1],

```

```

        long procrec[MAXPROCS+1]);

/* Determines the maximum resource utilization of links or
   processors */

long findMax (long linkUtil[MAXPROCS+1][MAXPROCS+1],
              long procUtil[MAXPROCS+1]);

/* Updates resource utilization tables given assignment of a node to a
   particular processor. */

void updateTables (long LUtil[MAXPROCS+1][MAXPROCS+1],
                  long PUtil[MAXPROCS+1], int *edgemat,
                  int *routes[MAXPROCS+1][MAXPROCS+1],
                  int procAssigned, int readyTask, int nodes,
                  struct schedData *sched);

/* Assigns appropriate overhead time for message passing to the node */

void setOverhead (int *edgemat, int *routes[MAXPROCS+1][MAXPROCS+1],
                 int procAssigned, int readyTask, int nodes,
                 struct schedData *sched, int *rDelay);

/* Recursively determines the indices for each node in the graph
   according to precedences and communication contention */

void assignIndex (struct schedData *sched, int *edgemat, int nodes,
                 long Ctime, int current, int index);

/* Sorts the resource utilization amounts in decreasing order */

void sortUtil(long tempLinkUtil[MAXPROCS+1][MAXPROCS+1],
              long tempProcUtil[MAXPROCS+1],
              long utilList[MAXPROCS+(MAXPROCS*MAXPROCS)]);

#endif

```

C. SCHEDULE FUNCTIONS SOURCE FILE

```

/* This file is schedule.c. It is the source file for all RPS scheduling
   functions */

#include "schedule.h"
#include <stdio.h>
#include <limits.h>

FILE *graphFile;
FILE *schedFile;
FILE *routing;

```

```

void generateSchedule ()
{
    int      *edgematrix;
    int      i, j, numnodes, numedges, numprocs;
    int      *routes[MAXPROCS+1][MAXPROCS+1];
    int      *routeDelay;

    struct schedData  *schedule;

    graphFile = fopen("graph.cgd", "r");
    fscanf (graphFile, "%d%d\n", &numnodes, &numedges);
    routing = fopen("routing.cgd", "r");
    fscanf (routing, "%d\n", &numprocs);

    for (i=0; i<MAXPROCS+1; i++) {          /* initialize routing
                                           matrix pointers      */
        for (j=0; j<MAXPROCS+1; j++) {      /* to NULL          */
            routes[i][j] = NULL;
        }
    }

    edgematrix = (int*) calloc (numnodes*numnodes, sizeof(int));

    for (i=0; i<numnodes*numnodes; i++) {   /* initialize edge matrix */
        *(edgematrix+i) = 0;                 /* times to zero          */
    }

    schedule = (struct schedData*) calloc ((numnodes),
                                           sizeof(struct schedData));

    readGraph (edgematrix, schedule, numnodes);
    readRouting (routes, numprocs);

    routeDelay = (int*) calloc ((numprocs), sizeof(int));

    for (i=0; i<numprocs; i++) {
        *(routeDelay+i) = 0;
    }

    PSHeuristic (routes, edgematrix, schedule, numprocs, numnodes,
                 routeDelay);

    outputSchedule (schedule, numnodes, numprocs, routeDelay);

    /* free all dynamically allocated memory */

    free((void*) routeDelay);
    free((void*) edgematrix);
    free((void*) schedule);
    for (i=0; i<MAXPROCS+1; i++) {
        for (j=0; j<MAXPROCS+1; j++) {

```



```

        if (routes[i][j] != NULL) {
            free((void*) routes[i][j]);
        }
    }
}

void
readGraph (int *edgemat, struct schedData *sched, int nodes)
{
    int i, j, number, from, to, pred, succ;
    char junk[80];

    for (i=0; i<nodes; i++) {
        fscanf (graphFile, "%d\n", &number);          /* get node number */
        fscanf (graphFile, "%s\n", junk);
        fscanf (graphFile, "%s\n", junk);

        /* store nodes computation time in comp time array */
        fscanf (graphFile, "%d\n", &(sched+number)->compTime);

        (sched+number)->processor = -1;
        (sched+number)->overheadTime = 0;
        (sched+number)->commTime = 0;
        (sched+number)->index = INT_MAX;

        fscanf (graphFile, "%d", &from);                /* get number of
                                                         predecessors */

        for (j=0; j<from; j++) {                        /* get predecessor
                                                         and edge */
            fscanf (graphFile, "%d", &pred);            /* communication time */
            fscanf (graphFile, "%d", edgemat+pred+(number*nodes));
        }
        fscanf (graphFile, "\n");

        fscanf (graphFile, "%d", &to);                /* get number of succesors */

        for (j=0; j<to; j++) {                          /* get succesor and edge */
            fscanf (graphFile, "%d", &succ);            /* communication time */
            fscanf (graphFile, "%d", edgemat+number+(succ*nodes));
        }
        fscanf (graphFile, "\n\n");
    }
    fclose(graphFile);
}

void
readRouting (int *routes[MAXPROCS+1][MAXPROCS+1], int procs)

```

```

{
    int i, j, to, from, links;

    for (i=0; i<procs*(procs-1); i++) {
        fscanf (routing, "%d%d%d\n", &from, &to, &links); /* get to and
                                                             from and
                                                             number of
                                                             links      */
        routes[from][to] = (int*) calloc (links, sizeof(int));
        *routes[from][to] = links; /* store # of
                                     links      */

        for (j=1; j<links; j++) { /* get all processors in route */
            fscanf (routing, "%d", routes[from][to]+j);
        }
        fscanf (routing, "\n");
    }
    fclose(routing);

    for (i=0; i<procs; i++) { /* ensure that a route exists between */
        for (j=0; j<procs; j++) { /* all processors */
            if (i != j) {
                if (routes[i][j] == NULL) {
                    printf ("Routing file is incomplete.");
                    exit (0);
                }
            }
        }
    }

    /* figure routes from processors to host */
    for (i=0; i<procs; i++) {
        routes[procs][i] = (int*)calloc((*routes[0][i])+1, sizeof(int));
        *routes[procs][i] = (*routes[0][i])+1;
        if (*routes[procs][i] > 1) {
            *(routes[procs][i]+1) = 0;
        }
        for (j=2; j<*routes[procs][i]; j++) {
            *(routes[procs][i]+j) = *(routes[0][i]+j-1);
        }
    }

    /* figure routes from host to processors */
    for (i=0; i<procs; i++) {
        routes[i][procs] = (int*)calloc((*routes[i][0])+1, sizeof(int));
        *routes[i][procs] = (*routes[i][0])+1;
        for (j=1; j<(*routes[i][procs])-1; j++) {
            *(routes[i][procs]+j) = *(routes[i][0]+j);
        }
        if (*routes[i][procs] > 1) {
            *(routes[i][procs]+(*routes[i][procs])-1) = 0;
        }
    }
}

```

```

    }
}

void
outputSchedule (struct schedData *sched, int nodes, int procs,
                int *rDelay)
{
    /* output all schedule data in form that can be read by the code
       generator and output any additional data that the is useful to
       the user */

    int i, j;
    long temp, total=0, largest=0;

    schedFile = fopen("schedule.cgd", "w");

    fprintf (schedFile, "Number of processors:%d\n\n", procs);

    for (i=1; i<nodes; i++) {
        fprintf (schedFile, "Node %d : processor %d, order %d\n", i,
                (sched+i)->processor, (sched+i)->order);
    }
    fprintf (schedFile, "\n");
    for (i=1; i<nodes; i++) {
        fprintf (schedFile, "Node %d index: %d\n", i, (sched+i)->index);
    }
    fprintf (schedFile, "\n");

    for (i=1; i<nodes; i++) {
        fprintf (schedFile, "Node %d computation time : %d", i,
                (sched+i)->compTime);
        fprintf (schedFile, "      overhead time : %d\n",
                (sched+i)->overheadTime);
    }
    fprintf (schedFile, "\n");

    for (i=0; i<procs; i++) {
        temp = 0;
        for (j=1; j<nodes; j++) {
            if ((sched+j)->processor == i) {
                temp += ((long)(sched+j)->compTime +
                        (long)(sched+j)->overheadTime);
            }
        }
        if (largest < ((long)*(rDelay+i)+temp)) {
            largest = (long)*(rDelay+i)+temp;
        }
        fprintf (schedFile, "Processor %d routing delay : %d", i,
                *(rDelay+i));
    }
}

```

```

        fprintf (schedFile, "      Total time : %ld\n",
                (long)*(rDelay+i)+temp);
    }

    for (i=1; i<nodes; i++) {
        total += (long)(sched+i)->compTime;
    }

    fprintf (schedFile, "\nExpected execution time: %ld\n", largest);
    fprintf (schedFile, "Total sequential execution time: %ld\n", total);
    fprintf (schedFile, "Expected speedup: %3f\n",
            ((float)total/(float)largest));

    fclose(schedFile);
}

void
PSHeuristic (int *routes[MAXPROCS+1][MAXPROCS+1], int *edgemat,
            struct schedData *sched, int procs, int nodes,
            int *routeDelay)
{
    long    linkUtil[MAXPROCS+1][MAXPROCS+1];
    long    procUtil[MAXPROCS+1];
    long    tempLinkUtil[MAXPROCS+1][MAXPROCS+1];
    long    tempProcUtil[MAXPROCS+1];
    long    tempUtilList[MAXPROCS+(MAXPROCS*MAXPROCS)];
    long    bestUtilList[MAXPROCS+(MAXPROCS*MAXPROCS)];
    int     *order;      /* array that hold the order that nodes are to be
                        scheduled in */
    int     i, j, k, readyTask, currentProc, selection, count,
            bestfound, ord;
    long    cylTime, maxUtil, temp, finishTime, tempCylTime;
    int     minIndex=0;

    for (i=0; i<=MAXPROCS; i++) {      /* initialize processor and link */
        procUtil[i] = 0;                /* utilization tables to zero */
        for (j=0; j<=MAXPROCS; j++) {
            linkUtil[i][j] = 0;
        }
    }

    order = (int*) calloc ((nodes-1), sizeof(int));

    determineOrder (order, edgemat, sched, nodes); /* determine order to
                                                    schedule nodes in */

    sched->processor = procs;

    printf("Do you wish first best (0), or best overall(1) scheduling?");
    scanf("%d", &selection);
    printf("\n");
}

```

```

for (i=0; i<nodes-1; i++) {
    maxUtil = LONG_MAX;

    for (j=0; j<MAXPROCS; j++) {
        bestUtilList[j] = LONG_MAX;
        for (k=0; k<MAXPROCS; k++) {
            bestUtilList[MAXPROCS+j+(k*MAXPROCS)] = LONG_MAX;
        }
    }

    readyTask = *(order+i);          /* get ready task from order list */
    for (j=0; j<procs; j++) {

        /* set temporary utilization tables to current schedule */
        copyTables (linkUtil, procUtil, tempLinkUtil, tempProcUtil);

        /* update tables to reflect readyTask being schedule on
           processor j */
        updateTables(tempLinkUtil, tempProcUtil, edgemat, routes, j,
                     readyTask, nodes, sched);

        if (!selection) {
            temp = findMax(tempLinkUtil, tempProcUtil); /* get maximum
                                                         resource
                                                         utilization */
            if (temp < maxUtil) { /* if this is the minimum so far, */
                maxUtil = temp; /* record max utilization and the */
                currentProc = j; /* processor */
            }
        }
        else {
            /* order Util figures */
            sortUtil(tempLinkUtil, tempProcUtil, tempUtilList);
            count = bestfound = 0;
            while (count<MAXPROCS+(MAXPROCS*MAXPROCS) && !bestfound) {
                /* if new assignment is better, keep it */
                if (tempUtilList[count] < bestUtilList[count]) {
                    currentProc = j;
                    for (k=0; k<MAXPROCS+(MAXPROCS*MAXPROCS); k++) {
                        bestUtilList[k] = tempUtilList[k];
                    }
                    bestfound = 1;
                }
                /* if new assignment is worse, keep old assignment */
                if (tempUtilList[count] > bestUtilList[count]) {
                    bestfound = 1;
                }
                count++;
            }
        }
    }
}

```

```

    }

    /* update tables to reflect readyTask being schedule on
       currentProc */
    updateTables(linkUtil, procUtil, edgemat, routes, currentProc,
                 readyTask, nodes, sched);

    /* assign readyTask to currentProc */
    (sched+readyTask) -> processor = currentProc;

    setOverhead(edgemat, routes, currentProc, readyTask, nodes,
                 sched, routeDelay);
}

/* determine length of the cylinder by adding comptime, overhead, and
   routing on each processor and comparing to find the largest */
cylTime = 0;
for (i=0; i<procs; i++) {
    tempCylTime = (long)*(routeDelay+i);
    for (j=1; j<nodes; j++) {
        if ((sched+j)->processor == i) {
            tempCylTime += ((long)(sched+j)->compTime
                           + (long)(sched+j)->overheadTime);
        }
        if (tempCylTime > cylTime) {
            cylTime = tempCylTime;
        }
    }
}

/* set start times, finish times, and order of execution */
for (i=0; i<procs; i++) {
    finishTime = 0;
    ord = 1;
    for (j=1; j<nodes; j++) {
        if ((sched+j)->processor == i) {
            (sched+j)->order = ord
            (sched+j)->startTime = finishTime;
            finishTime += (long)(sched+j)->compTime
                           + (long)(sched+j)->overheadTime;
            (sched+j)->finishTime = finishTime
                                   + (long)(sched+j)->commTime+(long)*(routeDelay+i);
            ord++;
        }
    }
}

/* find indicies */
assignIndex (sched, edgemat, nodes, cylTime, 1, minIndex);

```

```

/* adjust indices so minimum index is zero */
for (i=1; i<nodes; i++) {
    if ((sched+i)->index < minIndex) {
        minIndex = (sched+i)->index;
    }
}
for (i=1; i<nodes; i++) {
    (sched+i)->index -= minIndex;
}

free((void*) order);
}

void
determineOrder (int *ord, int *edgemat, struct schedData *sched, int
nodes)
{
    int i, j, k, bytes;
    int *scheduled;
    long temp, max;

    scheduled = (int*) calloc (nodes, sizeof(int));

    for (i=1; i<nodes; i++) { /* indicate that no nodes are ready to */
        *(scheduled+i) = 0; /* be scheduled (0) */
    }

    *scheduled = 1; /* indicate that host node is already
                    scheduled (1) */

    for (i=0; i<nodes-1; i++) {
        max = 0;
        for (j=1; j<nodes; j++) { /* check to see which node has
                                   greatest */
            if (*(scheduled+j)) { /* comptime plus message passing
                                   time */
                temp = (long)(sched+j)->compTime;
                for (k=0; k<nodes; k++) {
                    if (*(edgemat+k+(nodes*j)) != 0) {
                        temp += (long)(INST_OVHD+SETUP);
                    }
                    if (*(edgemat+j+(nodes*k)) != 0) {
                        bytes = *(edgemat+j+(nodes*k));
                        temp += (long)INST_OVHD;
                        temp += (((long)SETUP*(long)((bytes/BLOCK_SIZE)+1))
                                + (long)(BYTE1*(float)bytes));
                    }
                }
                if (temp > max) {
                    max = temp;
                }
            }
        }
    }
}

```

```

        *(ord+i) = j;
    }
}

*(scheduled+(*(ord+i))) = 1; /* indicate that it is scheduled and
                             repeat until all nodes are
                             scheduled */
}

free((void*) scheduled);
}

void
copyTables (long linksrc[MAXPROCS+1][MAXPROCS+1],
            long procsrc[MAXPROCS+1],
            long linkrec[MAXPROCS+1][MAXPROCS+1],
            long procrec[MAXPROCS+1])
{
    int i, j;

    /* copy one link utilization and processor utilization table to the
       others */

    for (i=0; i<=MAXPROCS; i++) {
        procrec[i] = procsrc[i];
        for (j=0; j<=MAXPROCS; j++) {
            linkrec[i][j] = linksrc[i][j];
        }
    }
}

long
findMax (long linkUtil[MAXPROCS+1][MAXPROCS+1],
         long procUtil[MAXPROCS+1])
{
    int i, j;
    long temp=0;

    for (i=0; i<MAXPROCS; i++) { /* find the largest utilization time */
        if (procUtil[i] > temp) { /* of all processors */
            temp = procUtil[i];
        }
    }

    for (i=0; i<MAXPROCS; i++) { /* find largest utilization time of */
        for (j=0; j<MAXPROCS; j++) { /* all links */
            if (linkUtil[i][j] > temp) {
                temp = linkUtil[i][j];
            }
        }
    }
}

```



```

    }
}
return temp; /* return the largest utilization time of processors
              and links */
}

void
updateTables (long LUtil[MAXPROCS+1][MAXPROCS+1],
              long PUtil[MAXPROCS+1], int *edgemat,
              int *routes[MAXPROCS+1][MAXPROCS+1], int procAssigned,
              int readyTask, int nodes, struct schedData *sched)
{
    int i, j, numlinks, from, to, bytes;
    int *currentRoute;

    /* add comp time to proc utilization table for proc assigned */
    PUtil[procAssigned] += (long)(sched+readyTask)->compTime;

    /* for all previously scheduled parents of the node */
    for (i=0; i<nodes; i++) {
        if (*(edgemat+i+(nodes*readyTask)) != 0
            && (sched+i)->processor != -1) {
            bytes = *(edgemat+i+(nodes*readyTask));
            from = (sched+i) -> processor;

            /* if on the same processor, add memory access time to
               processor utilization */
            if (from == procAssigned) {
                PUtil[from] += ((long)MEM_ACCESS +
                               (long)((float)bytes*BYTE_MEM_ACCESS));
                PUtil[procAssigned] += ((long)MEM_ACCESS +
                                         (long)((float)bytes*BYTE_MEM_ACCESS));
            }
            /* otherwise, add overhead, setup, and first link time to
               processor utilization */
            else {
                PUtil[from] += (long)INST_OVHD;
                PUtil[procAssigned] += (long)(INST_OVHD+SETUP);
                currentRoute = routes[from][procAssigned];
                numlinks = *currentRoute;
                PUtil[from] += (((long)SETUP*(long)((bytes/BLOCK_SIZE)+1))
                               + (long)(BYTE1*(float)bytes));

                /* if greater than 1 link and greater than 1 block, add
                   extra link time to processor utilization */
                if (numlinks > 1 && bytes > BLOCK_SIZE) {
                    PUtil[from] += ((long)ROUTING +
                                     (long)(BYTE2*(float)BLOCK_SIZE) +
                                     (long)(BYTE3*(float)((bytes/BLOCK_SIZE)

```

```

-1)*BLOCK_SIZE));
}

/* for each link along the route, add comm time to link utilization */
for (j=1; j<numlinks; j++) {
    if (j==1) {
        LUtil[from][*(currentRoute+j)] +=
            (((long)SETUP*(long)((bytes/BLOCK_SIZE)+1))
            + (long)(BYTE1*(float)bytes));
    }
    else {
        if (bytes <= BLOCK_SIZE) {
            LUtil[from][*(currentRoute+j)] += ((long)ROUTING +
            (long)(BYTE2*(float)bytes));
        }
        else {
            LUtil[from][*(currentRoute+j)] += ((long)ROUTING +
            (long)(BYTE2*(float)BLOCK_SIZE)
            + (long)(BYTE3*(float)(bytes
            -BLOCK_SIZE)));
        }
    }
    from = *(currentRoute+j);
    PUtil[from] += ((long)RTNG_DLY +
    (long)(BYTE_RT_DLY*(float)bytes));
}

if (numlinks==1) {
    LUtil[from][procAssigned] += (((long)SETUP*(long)((bytes/
    BLOCK_SIZE)+1))+(long)(BYTE1*(float)bytes));
}
else {
    if (bytes <= BLOCK_SIZE) {
        LUtil[from][procAssigned] += ((long)ROUTING +
        (long)(BYTE2*(float)bytes));
    }
    else {
        LUtil[from][procAssigned] += ((long)ROUTING +
        (long)(BYTE2*(float)BLOCK_SIZE) +
        (long)(BYTE3*(float)(bytes-BLOCK_SIZE)));
    }
}
}
}

/* for all previously scheduled children of the node */
for (i=0; i<nodes; i++) {
    if (*(edgemat+readyTask+(nodes*i)) != 0
        && (sched+i)->processor != -1) {

```

```

bytes = *(edgemat+readyTask+(nodes*i));
to = (sched+i) -> processor;
from = procAssigned;

/* if on the same processor, add memory access time to
processor utilization */
if (to == procAssigned) {
    PUtil[to] += ((long)MEM_ACCESS +
                  (long)((float)bytes*BYTE_MEM_ACCESS));
    PUtil[procAssigned] += ((long)MEM_ACCESS +
                            (long)((float)bytes*BYTE_MEM_ACCESS));
}
/* otherwise, add overhead, setup, and first link time to
processor utilization */
else {
    PUtil[from] += (long)INST_OVHD;
    PUtil[to] += (long)(INST_OVHD+SETUP);
    currentRoute = routes[from][to];
    numlinks = *currentRoute;
    PUtil[from] += (((long)SETUP*(long)((bytes/BLOCK_SIZE)+1))
                  + (long)(BYTE1*(float)bytes));

    /* if greater than 1 link and greater than 1 block, add
    extra link time to processor utilization */
    if (numlinks > 1 && bytes > BLOCK_SIZE) {
        PUtil[from] += ((long)ROUTING +
                        (long)(BYTE2*(float)BLOCK_SIZE) +
                        (long)(BYTE3*(float)((bytes/BLOCK_SIZE)
                        -1)*BLOCK_SIZE));
    }

    /* for each link along the route, add comm time to link
    utilization */
    for (j=1; j<numlinks; j++) {
        if (j==1) {
            LUtil[from][*(currentRoute+j)] +=
                (((long)SETUP*(long)((bytes/BLOCK_SIZE)+1))
                + (long)(BYTE1*(float)bytes));
        }
        else {
            if (bytes <= BLOCK_SIZE) {
                LUtil[from][*(currentRoute+j)] += ((long)ROUTING +
                                                    (long)(BYTE2*(float)bytes));
            }
            else {
                LUtil[from][*(currentRoute+j)] += ((long)ROUTING +
                                                    (long)(BYTE2*(float)BLOCK_SIZE) +
                                                    (long)(BYTE3*(float)(bytes-BLOCK_SIZE)));
            }
        }
        from = *(currentRoute+j);
    }
}

```



```

currentRoute = routes[from][procAssigned];
numlinks = *currentRoute;
(sched+i)->overheadTime += ((SETUP*((bytes/BLOCK_SIZE)+1))
                           + (int)(BYTE1*(float)bytes));

if (numlinks > 1 && bytes > BLOCK_SIZE) {
    (sched+i)->overheadTime += (ROUTING +
                              (int)(BYTE2*(float)BLOCK_SIZE) +
                              (int)(BYTE3*(float)((bytes/BLOCK_SIZE)
                                                  -1)*BLOCK_SIZE));
    temp = ROUTING + (int)(BYTE3*(float)(bytes % BLOCK_SIZE))
            + ((ROUTING + (int)(BYTE2*(float)(BLOCK_SIZE)))
              * (numlinks - 2));
    if (temp > (sched+i)->commTime) {
        (sched+i)->commTime = temp;
    }
}
if (numlinks > 1 && bytes <= BLOCK_SIZE) {
    temp = (ROUTING + (int)(BYTE2*(float)bytes))
            * (numlinks - 1);
    if (temp > (sched+i)->commTime) {
        (sched+i)->commTime = temp;
    }
}

/* add routing delay to delay array for all processors that
   messages pass through */
for (j=1; j<numlinks; j++) {
    from = *(currentRoute+j);
    *(rDelay+from) += (RTNG_DLY +
                      (int)(BYTE_RT_DLY*(float)bytes));
}
}
}

/* for all previously scheduled children of the node */
for (i=0; i<nodes; i++) {
    if (*(edgemat+readyTask+(nodes*i)) != 0
        && (sched+i)->processor != -1) {
        bytes = *(edgemat+readyTask+(nodes*i));
        to = (sched+i) -> processor;
        from = procAssigned;
        /* if on the same processor, add memory access time to
           overhead */
        if (to == procAssigned) {
            (sched+i)->overheadTime += MEM_ACCESS;
            (sched+readyTask)->overheadTime += MEM_ACCESS;
        }
        /* otherwise, add message passing time, based on number of
           blocks and number of bytes, to overhead */
    }
}

```



```

        for (j=0; j<MAXPROCS; j++) {
            utilList[MAXPROCS+i+(j*MAXPROCS)] = tempLinkUtil[i][j];
        }
    }

    /* sort the utilList, largest first */
    for (i=0; i<index; i++) {
        for (j=0; j<index; j++) {
            if (utilList[j] < utilList[j+1]) {
                temp = utilList[j];
                utilList[j] = utilList[j+1];
                utilList[j+1] = temp;
            }
        }
    }
}

```

D. PACKAGING HEADER FILE

```

/* This file is codegen.h. It is the header file for the RPS Packaging
   functions */

#ifndef __CODEGEN_H
#define __CODEGEN_H

struct node {          /* structure to store all needed data about the */
    int    number      /* nodes of the task graph */
    char   filename[9];
    char   nodename[20];
    int    numfrom;
    int    numto;
    int    *from;
    int    *to;
    int    index;
};

/* Function that generates the code that is run on the host PC and the
   transputers. Also generates a batch file to compile all code
   generated */

int codeGen();

/* Creates the files for the code generation and calls all functions
   that generate code */

void createProcesses(int procs, int nodes, int edges);

/* Reads the schedule file and stores the processor mapping in map and
   the run order in ord */

```

```

void readSchedule (int *map, int *ord, int nodes);

/* Reads the graph file and stores all of the nodes data in theNodes */

void readNodes (struct node *theNodes, int nodes);

/* Generates a list of all edges in the graph and the sending and
recieving nodes associated with them */

void generateEdges (struct node *theNodes, int *edgeMatrix, int *map,
                    int nodes);

/* Generates the code for any processor based on the nodes assigned to
it, The location of the other nodes, and the edges of the graph */

void makeProc (int *map, int *ord, int *edgeMat, struct node *theNodes,
               int nodes, int edges, int pnum, int time, int synch);

/* Generates the code for the host PC processor based on the I/O routine
provided and the location of nodes that require I/O */

void makeHost (int *map, int *edgeMat, struct node *theNodes, int nodes,
               int edges, int nprocs, int time);

#endif

```

E. PACKAGING SOURCE FILE

```

/* This file is codegen.c. It is the source file for all RPS Packager
functions */

#define MAXPROCS 16

#include "codegen.h"
#include <stdio.h>
#include <limits.h>

char *incSuff = ".inc";
char *toSuff = ".to";
char *fileSuff = ".fil";
char *codeSuff = ".tcs";

FILE *schedule;
FILE *graph;
FILE *process;

int
codegen()
{
    int i, numprocs, numnodes, numedges;

```

```

schedule = fopen("schedule.cgd", "r");
graph = fopen("graph.cgd", "r");

fscanf (schedule, "Number of processors:%d\n\n", &numprocs);
fscanf (graph, "%d%d\n", &numnodes, &numedges);

if (numprocs<=MAXPROCS) {
    createProcesses(numprocs, numnodes, numedges);
}
else {
    if (numprocs>MAXPROCS)
        printf(
            "This program cannot be used with more than %d processors.\n"
            ,MAXPROCS);
}

fclose(schedule);
fclose(graph);

return 0;
}

void
createProcesses(int procs, int nodes, int edges)
{
    char processName[9] = "host.c";
    int i, temp, time, synch;
    int *mapping;
    int *order;
    int *edgeMatrix;
    struct node *theNodes;
    FILE *batfile;

    mapping = (int*) calloc ((nodes-1), sizeof(int));
    order = (int*) calloc ((nodes-1), sizeof(int));
    theNodes = (struct node*) calloc (nodes, sizeof(struct node));
    edgeMatrix = (int*) calloc ((4*edges), sizeof(int));

    readSchedule (mapping, order, nodes);
    readNodes (theNodes, nodes);

    generateEdges (theNodes, edgeMatrix, mapping, nodes);

    printf
        ("Do you wish time measurement to be included in your program?\n");
    printf ("(0 for no, 1 for yes):");
    scanf ("%d", &time);

    printf
        ("Do you wish synchronization to be included in your program?\n");
    printf ("(0 for no, 1 for yes):");

```

```

scanf ("%d", &synch);

/* create batch file for compiling program */
batfile = fopen("compile.bat", "w");

fprintf (batfile,
        "c:\\tc\\bin\\tcc -ml -c -Ic:\\parasoft\\hostinc host.c\n");
fprintf (batfile, "c:\\tc\\bin\\tcc -ml -ehost.exe host.obj
        c:\\parasoft\\lib\\exptc.lib\n");
process = fopen(processName, "w");
makeHost (mapping, edgeMatrix, theNodes, nodes, edges, procs, time);
fclose(process);

processName[0] = 'p';
processName[1] = 'r';
processName[2] = 'o';
processName[3] = 'c';

for (i=0; i<procs; i++) {
    if (i<10) {
        processName[4] = 48+i;
        processName[5] = '.';
        processName[6] = 'c';
        processName[7] = '\\0';
    }
    else {
        if (i<100) {
            temp = i;
            temp = temp/i;
            processName[4] = 48+temp;
            temp = i-temp*10;
            processName[5] = 48+temp;
            processName[6] = '.';
            processName[7] = 'c';
            processName[8] = '\\0';
        }
    }

    fprintf (batfile, "c:\\parasoft\\bin\\tcc -o proc%d %s\n"
            , i, processName);
    process = fopen(processName, "w");
    makeProc (mapping, order, edgeMatrix, theNodes, nodes, edges,
            i, time, synch);
    fclose(process);
}

free((void*) mapping);
free((void*) edgeMatrix);
for (i=0; i<nodes; i++) {
    free((void*) (theNodes+i->from);
    free((void*) (theNodes+i->to);

```

```

    }
    free((void*) theNodes);
}

void
readSchedule (int *map, int *ord, int nodes)
{
    int i, trash;

    for (i=1; i<nodes; i++,map++,ord++) {
        fscanf (schedule, "Node %d : processor %d, order %d\n", &trash,
                map, ord);
    }
    fscanf (schedule, "\n");
}

void
readNodes (struct node *theNodes, int nodes)
{
    int i, j, trash;
    float junk;

    for (i=0; i<nodes; i++, theNodes++) {
        fscanf (graph, "%d\n", &theNodes->number);
        fscanf (graph, "%s\n", theNodes->filename);
        fscanf (graph, "%s\n", theNodes->nodename);
        fscanf (graph, "%f\n", &junk);
        fscanf (graph, "%d", &theNodes->numfrom);

        /* allocate memory for parents of node */
        if (theNodes->numfrom > 0)
            theNodes->from = (int*)calloc (theNodes->numfrom, sizeof(int));
        else
            theNodes->from = 0;

        /* allocate memory for children of node */
        for (j=0; j<theNodes->numfrom; j++) {
            fscanf (graph, "%d", (theNodes->from+j));
            fscanf (graph, "%f", &junk);
        }
        fscanf (graph, "\n");

        fscanf (graph, "%d", &theNodes->numto);

        if (theNodes->numto > 0)
            theNodes->to = (int*) calloc (theNodes->numto, sizeof(int));
        else
            theNodes->to = 0;

        for (j=0; j<theNodes->numto; j++) {

```

```

        fscanf (graph, "%d", (theNodes->to+j));
        fscanf (graph, "%f", &junk);
    }
    fscanf (graph, "\n\n");

    if (i)
        fscanf (schedule, "Node %d index: %d\n", &trash,
                &theNodes->index);
}

void
generateEdges (struct node *theNodes, int *edgeMatrix, int *map,
               int nodes)
{
    int i, j, k, buffnum = 0;

    for (i=0; i<nodes; i++, theNodes++) {
        for (j=0; j<theNodes->numto; j++, edgeMatrix +=4) {
            *edgeMatrix = theNodes->number;          /* set source of edge */
            *(edgeMatrix+1) = *(theNodes->to+j);      /* set destination of
                                                         edge */

            /* set the edge number */
            if (theNodes->number) {
                if (*(map+i-1) == *(map + (*(theNodes->to+j)) - 1)) {
                    buffnum++;
                    *(edgeMatrix+2) = buffnum;
                }
                else {
                    *(edgeMatrix+2) = 0;
                }
            }
        }
    }
}

void
makeProc (int *map, int *ord, int *edgeMat, struct node *theNodes,
          int nodes, int edges, int pnum, int time, int synch)
{
    int i, j, k, count, diff, numincludes, maxIndex=0, minIndex=INT_MAX;
    int tocounter, fromcounter, currentProc, overallmax=0,
        overallmin=INT_MAX;
    int currentEdge, arraycount;
    FILE *nodefile;
    char buffer[81];
    char currentFile[13];
    char arraybuf[10];

    fprintf (process, "#include \"express.h\"\n");

```

```

/* add other includes to the file */

for (i=1; i<nodes; i++) {
    if (*(map+i-1) == pnum) {
        strcpy (currentFile, (theNodes+i)->filename);
        strcat (currentFile, incSuff);
        nodefile = fopen(currentFile, "r");
        fscanf (nodefile, "%d\n", &numincludes);
        for (j=0; j<numincludes; j++) {
            fscanf (nodefile, "%s\n", buffer);
            fprintf (process, "#include %s\n", buffer);
        }
        fclose(nodefile);
    }
}
fprintf (process, "\n");

/* declare mapping variable */
fprintf (process, "int mapping[%d] = {0", nodes);
for (i=0; i<nodes-1; i++)
    fprintf (process, ", %d", *(map+i));
fprintf (process, "};\n");

/* declare types variable */
fprintf (process, "int types[%d] = {0", edges+1);
for (i=1; i<=edges; i++)
    fprintf (process, ", %d", i);
fprintf (process, "};\n\n");

/* compute maximum and minimum indices */
for (i=1; i<nodes; i++) {
    if (*(map+i-1) == pnum) && ((theNodes+i)->index > maxIndex)) {
        maxIndex = (theNodes+i)->index;
    }
    if (*(map+i-1) == pnum) && ((theNodes+i)->index < minIndex)) {
        minIndex = (theNodes+i)->index;
    }
    if ((theNodes+i)->index > overallmax) {
        overallmax = (theNodes+i)->index;
    }
    if ((theNodes+i)->index < overallmin) {
        overallmin = (theNodes+i)->index;
    }
}
if (minIndex == INT_MAX) {
    minIndex = 0;
}
diff = maxIndex - minIndex;

/* determine if same process buffering is needed and which edges of

```

```

the graph are involved
for (i=1; i<nodes; i++) {
    if (*(map+i-1) == pnum) {
        strcpy (currentFile, (theNodes+i)->filename);
        strcat (currentFile, toSuff);
        nodefile = fopen(currentFile, "r");

        for (j=0; j<(theNodes+i)->numto; j++) {
            fgets (buffer, 80, nodefile);
            currentProc = *((theNodes+i)->to+j);
            for (k=0; k<edges; k++) {
                if ((theNodes+i)->number == *(edgeMat+(4*k))
                    && currentProc == *(edgeMat+(4*k)+1)
                    && *(edgeMat+(4*k)+2)) {
                    count = 0;
                    while (buffer[count] != '\n' && buffer[count] != '[') {
                        fprintf (process, "%c", buffer[count]);
                        count++;
                    }
                    fprintf (process, "   buf%d[%d]", *(edgeMat+(4*k)+2),
                        diff+1);

                    if (buffer[count] == '[') {
                        arraycount = 0;
                        while (buffer[count+1+arraycount] != ']') {
                            arraybuf[arraycount] =
                                buffer[count+1+arraycount];
                            arraycount++;
                        }
                        arraybuf[arraycount] = '\0';
                    }
                    else {
                        arraybuf[0] = '1';
                        arraybuf[1] = '\0';
                    }
                    *(edgeMat+(4*k)+3) = atoi(arraybuf);

                    while (buffer[count] != '\n') {
                        fprintf (process, "%c", buffer[count]);
                        count++;
                    }
                    fprintf (process, ";\n");
                }
            }
        }
        fclose(nodefile);
    }
}
fprintf (process, "\n");

/* node functions */

```



```

for (i=1; i<nodes; i++) {
    tocounter = fromcounter = 0;
    if (*(map+i-1) == pnun) {
        strcpy (currentFile, (theNodes+i)->filename);
        strcat (currentFile, codeSuff);
        nodefile = fopen(currentFile, "r");
        while (fgets(buffer, 80, nodefile) != 0) { /* get next line */
            j = 0; /* of .tcs file */
            while (buffer[j] == ' ') {
                j++;
            }
            /* if it's a read, generate exread code */
            if (buffer[j] == '+' && buffer[j+1] == '+'
                && buffer[j+2] == '+') {
                currentProc = *((theNodes+i)->from+fromcounter);
                for (k=0; k<edges; k++) {
                    if ((theNodes+i)->number == *(edgeMat+(4*k)+1) &&
                        currentProc == *(edgeMat+(4*k))) {
                        currentEdge = k;
                    }
                }

                if (*(edgeMat+(4*currentEdge)+2)) {
                    if (*(edgeMat+(4*currentEdge)+3) > 1) {
                        for (k=0; k<j; k++) {
                            fprintf (process, " ");
                        }
                        fprintf (process, "for (loopcount=0; loopcount<");
                        fprintf (process, "%d; ",
                            *(edgeMat+(4*currentEdge)+3));
                        fprintf (process, "loopcount++) {\n");

                        for (k=0; k<j+3; k++) {
                            fprintf (process, " ");
                        }

                        count = j + 3;
                        while (buffer[count] != '(') {
                            count++;
                        }
                        count++;
                        while (buffer[count] != ',') {
                            fprintf (process, "%c", buffer[count]);
                            count++;
                        }
                        fprintf (process, "[loopcount] = ");

                        fprintf (process, "buf%d[",
                            *(edgeMat+(4*currentEdge)+2));
                        fprintf (process, "(index %");

```

```

        fprintf (process, " %d)][", diff+1);
        fprintf (process, "loopcount);\n");

        for (k=0; k<j; k++) {
            fprintf (process, " ");
        }
        fprintf (process, "]\n");

        fromcounter++;
    }
    else {

        for (k=0; k<j; k++) {
            fprintf (process, " ");
        }
        fprintf (process, "*(");

        count = j + 3;
        while (buffer[count] != '(') {
            count++;
        }
        count++;
        while (buffer[count] != ',') {
            fprintf (process, "%c", buffer[count]);
            count++;
        }
        fprintf (process, ") = ");

        fprintf (process, "buf%d[",
                    *(edgeMat+(4*currentEdge)+2));
        fprintf (process, "(index %");
        fprintf (process, " %d)];\n", diff+1);

        fromcounter++;
    }
}
else {

    for (k=0; k<j; k++) {
        fprintf (process, " ");
    }
    fprintf (process, "exread (");
    count = j + 3;
    while (buffer[count] != '(') {
        count++;
    }
    count++;
    while (buffer[count] != ')') {
        buffer[count+1] != ';' ) {
            fprintf (process, "%c", buffer[count]);
            count++;
        }
    }
}

```

```

    }

    fprintf (process, ", &mapping[%d], &types[",
                                                    currentProc);
    fprintf (process, "%d]);\n", currentEdge);
    fromcounter++;
}

}
else {
    /* if it's a write, generate exwrite code */
    if (buffer[j] == '-' && buffer[j+1] == '-' &&
        buffer[j+2] == '-') {
        currentProc = *((theNodes+i)->to+tocounter);
        for (k=0; k<edges; k++) {
            if ((theNodes+i)->number == *(edgeMat+(4*k)) &&
                currentProc == *(edgeMat+(4*k+1))) {
                currentEdge = k;
            }
        }

        if (*(edgeMat+(4*currentEdge)+2)) {
            if (*(edgeMat+(4*currentEdge)+3) > 1) {
                for (k=0; k<j; k++) {
                    fprintf (process, " ");
                }
                fprintf (process,
                    "for (loopcount=0; loopcount<");
                fprintf (process, "%d; ",
                    *(edgeMat+(4*currentEdge)+3));
                fprintf (process, "loopcount++) {\n");

                for (k=0; k<j+3; k++) {
                    fprintf (process, " ");
                }
                fprintf (process, "buf%d[",
                    *(edgeMat+(4*currentEdge)+2));
                fprintf (process, "(index %");
                fprintf (process, " %d)][" , diff+1);
                fprintf (process, "loopcount] = ");

                count = j + 3;
                while (buffer[count] != '(') {
                    count++;
                }
                count++;
                while (buffer[count] != ',') {
                    fprintf (process, "%c", buffer[count]);
                    count++;
                }
                fprintf (process, "[loopcount];\n");
            }
        }
    }
}

```

```

        for (k=0; k<j; k++) {
            fprintf (process, " ");
        }
        fprintf (process, "\n");

        tocounter++;
    }
    else {
        for (k=0; k<j; k++) {
            fprintf (process, " ");
        }
        fprintf (process, "buf%d[",
                    *(edgeMat+(4*currentEdge)+2));
        fprintf (process, "(index %");
        fprintf (process, " %d)] = *(", diff+1);

        count = j + 3;
        while (buffer[count] != '(') {
            count++;
        }
        count++;
        while (buffer[count] != ',') {
            fprintf (process, "%c", buffer[count]);
            count++;
        }
        fprintf (process, ");\n");
        tocounter++;
    }
}
else {
    for (k=0; k<j; k++) {
        fprintf (process, " ");
    }
    fprintf (process, "exwrite (");
    count = j + 3;
    while (buffer[count] != '(') {
        count++;
    }
    count++;
    while (buffer[count] != ')') {
        while (buffer[count+1] != ';') {
            fprintf (process, "%c", buffer[count]);
            count++;
        }
    }

    fprintf (process, ", &mapping[%d], &types[",
                currentProc);
    fprintf (process, "%d]);\n", currentEdge);
    tocounter++;
}
}

```

```

        }
        /* otherwise, print the line to the file */
        else {
            fprintf (process, "%s", buffer);
        }
    }
}
fprintf (process, "\n");
fclose(nodefile);
}
fprintf (process, "\n");

/* generate main part of the node code */
fprintf (process, "int\nmain()\n{\n    int i, iterations;\n");

/* add if timing selected */
if (time) {
    fprintf (process, "    long timing[2];\n");
}

fprintf (process, "\n    mapping[0] = HOST;\n");

fprintf (process, "\n    exbroadcast(&iterations, mapping[0],
                                     sizeof(iterations),");
fprintf (process, "    ALLNODES, NULLPTR, &types[%d]);\n\n", edges);

fprintf (process, "    iterations -= %d;\n\n", diff);

/* add if timing selected */
if (time) {
    fprintf (process, "    timing[0] = exptime();\n\n");
}

/* add if synch selected */
if (synch) {
    for (i=0; i<overallmax-maxIndex; i++) {
        fprintf (process, "    exsync();\n");
    }
}
fprintf (process, "\n");

/* generate early cylinder iterations based on indices */
for (i=0; i<diff; i++) {
    for (j=1; j<nodes; j++) {
        for (k=1; k<nodes; k++) {
            if ((*map+k-1) == pnun) &&
                ((maxIndex - (theNodes+k)->index) <= i)
                && *(ord+k-1) == j) {
                fprintf (process, "    %s;\n", (theNodes+k)->nodename);
            }
        }
    }
}

```

```

    }
}
/* add if synch selected */
if (synch) {
    fprintf (process, "    exsync();\n");
}
fprintf (process, "\n");
}

fprintf (process, "    for (i=0; i<iterations; i++) {\n");

/* generated middle cylinder iterations */
for (i=1; i<nodes; i++) {
    for (j=1; j<nodes; j++) {
        if (*(map+j-1) == pnum && *(ord+j-1) == i) {
            fprintf (process, "        %s;\n", (theNodes+j)->nodename);
        }
    }
}
/* add if synch selected */
if (synch) {
    fprintf (process, "    exsync();\n");
}

fprintf (process, "    }\n\n");

/* generate final cylinder iterations based on indices */
for (i=0; i<diff; i++) {
    for (j=1; j<nodes; j++) {
        for (k=1; k<nodes; k++) {
            if ((*(map+k-1) == pnum) &&
                (((theNodes+k)->index - minIndex) < (diff-i))
                && *(ord+k-1) == j) {
                fprintf (process, "    %s;\n", (theNodes+k)->nodename);
            }
        }
    }
}
/* add if synch selected */
if (synch) {
    fprintf (process, "    exsync();\n");
}
fprintf (process, "\n");
}

/* add if synch selected */
if (synch) {
    for (i=0; i<minIndex-overallmin; i++) {
        fprintf (process, "    exsync();\n");
    }
}
}

```

```

fprintf (process, "\n");

/* add if timing selected */
if (time) {
    fprintf (process, "    timing[1] = extime();\n");
    fprintf (process, "    exwrite (timing, sizeof(timing),
                                   &mapping[0], &types[%d]);\n\n", edges);
}

fprintf (process, "    return 0;\n\n");
}

void
makeHost (int *map, int *edgeMat, struct node *theNodes, int nodes,
          int edges, int nprocs, int time)
{
    int i, j, k, count, tocounter, fromcounter, numincludes, current;
    int infiles, outfiles, overallmax = 0;
    FILE *nodefile;
    char buffer[81], buffer1[81];
    char currentFile[13];

    fprintf (process, "#include \"express.h\"\n");
    fprintf (process, "#include <stdio.h>\n");

    /* add other includes to the file */
    strcpy (currentFile, theNodes->filename);
    strcat (currentFile, incSuff);
    nodefile = fopen(currentFile, "r");
    fscanf (nodefile, "%d\n", &numincludes);
    for (j=0; j<numincludes; j++) {
        fscanf (nodefile, "%s\n", buffer);
        fprintf (process, "#include %s\n", buffer);
    }
    fprintf (process, "\n");
    fclose (nodefile);

    fprintf (process, "char *dev = \"/dev/transputer\";\n");

    for (i=0; i<nprocs; i++) {
        fprintf (process, "char *proc%d = \"proc%d\";\n", i, i);
    }
    fprintf (process, "\n");

    /* generate mapping variable */
    fprintf (process, "long  iterations;\n");
    fprintf (process, "int    i;\n");
    fprintf (process, "int    mapping[%d] = {0", nodes);
    for (i=0; i<nodes-1; i++)

```

```

    fprintf (process, ",%d", *(map+i));
    fprintf (process, "};\n");

    /* generate types variable */
    fprintf (process, "int    types[%d] = {0", edges+1);
    for (i=1; i<=edges; i++)
        fprintf (process, ",%d", i);
    fprintf (process, "};\n");

    strcpy (currentFile, theNodes->filename);
    strcat (currentFile, fileSuff);
    nodefile = fopen(currentFile, "r");

    /* generate file pointers for I/O */
    fscanf (nodefile, "%d\n", &infiles);
    for (j=1; j<=infiles; j++) {
        fscanf (nodefile, "%s\n", buffer);
        fprintf (process, "FILE  *infile%d;\n", j);
    }
    fscanf (nodefile, "%d\n", &outfiles);
    for (j=1; j<=outfiles; j++) {
        fscanf (nodefile, "%s\n", buffer);
        fprintf (process, "FILE  *outfile%d;\n", j);
    }
    fprintf (process, "\n");
    fclose (nodefile);

    tocounter = fromcounter = 0;

    strcpy (currentFile, theNodes->filename);
    strcat (currentFile, codeSuff);
    nodefile = fopen(currentFile, "r");

    for (i=1; i<nodes; i++) {
        if ((theNodes+i)->index > overallmax) {
            overallmax = (theNodes+i)->index;
        }
    }
    overallmax++;

    while (fgets(buffer, 80, nodefile) != 0) { /* get next line from
                                                file */
        j = 0;
        while (buffer[j] == ' ') {
            j++;
        }
        /* if it's a read, generate exread code */
        if (buffer[j] == '+' && buffer[j+1] == '+' && buffer[j+2] == '+'){
            current = *(theNodes->from+fromcounter);

            for (k=0; k<j; k++) {

```



```

        fprintf (process, " ");
    }
    fprintf (process, "if ((i >= %d) && ",
              (overallmax-(theNodes+current)->index));
    fprintf (process, "(i < iterations + %d)) {\n",
              (overallmax-(theNodes+current)->index));

    for (k=0; k<j+3; k++) {
        fprintf (process, " ");
    }
    fprintf (process, "exread (");
    count = j + 3;
    while (buffer[count] != '(') {
        count++;
    }
    count++;
    while (buffer[count] != ')' || buffer[count+1] != ';') {
        fprintf (process, "%c", buffer[count]);
        count++;
    }

    fprintf (process, ", &mapping[%d], &types[, current);
    for (k=0; k<edges; k++) {
        if (theNodes->number == *(edgeMat+(4*k)+1) &&
            current == *(edgeMat+(4*k))) {
            fprintf (process, "%d];\n", k);
        }
    }

    fgets(buffer, 80, nodefile);
    fprintf (process, " %s", buffer);

    for (k=0; k<j; k++) {
        fprintf (process, " ");
    }
    fprintf (process, ")\n");
    fromcounter++;
}

else {
    /* if it's a write, generate exwrite code */
    if (buffer[j] == '-' && buffer[j+1] == '-'
        && buffer[j+2] == '-') {
        current = *(theNodes->to+tocounter);

        for (k=0; k<j; k++) {
            fprintf (process, " ");
        }
        fprintf (process, "if ((i >= %d) && ",
                    (overallmax-1-(theNodes+current)->index));
        fprintf (process, "(i < iterations + %d)) {\n",

```

```

                                (overallmax-1-(theNodes+current)->index));

fgets(buffer1, 80, nodefile);
fprintf (process, "    %s", buffer1);

for (k=0; k<j+3; k++) {
    fprintf (process, " ");
}
fprintf (process, "exwrite (");
count = j + 3;
while (buffer[count] != '(') {
    count++;
}
count++;
while (buffer[count] != ')' || buffer[count+1] != ';') {
    fprintf (process, "%c", buffer[count]);
    count++;
}

fprintf (process, ", &mapping[%d], &types[, current);
for (k=0; k<edges; k++) {
    if (theNodes->number == *(edgeMat+(4*k)) &&
        current == *(edgeMat+(4*k)+1)) {
        fprintf (process, "%d]];\n", k);
    }
}
for (k=0; k<j; k++) {
    fprintf (process, " ");
}
fprintf (process, "]\n");
tocounter++;

}
else { /* otherwise, print the line to the file */
    fprintf (process, "%s", buffer);
}
}

}
fprintf (process, "\n");
fclose(nodefile);

/* generate main part of the host */
fprintf (process, "\nint\nmain()\n{\n");

if (time) {
    fprintf (process, "    long    times[2];\n");
}

fprintf (process, "    int    nprocs = %d, src = 0, fd;\n\n", nprocs);

fprintf (process, "    mapping[0] = HOST;\n\n");

```

```

strcpy (currentFile, theNodes->filename);
strcat (currentFile, fileSuff);
nodefile = fopen(currentFile, "r");
fscanf (nodefile, "%d\n", &infiles);
for (j=1; j<=infiles; j++) {
    fscanf (nodefile, "%s\n", buffer);
    fprintf (process, "    infile%d = fopen(\"%s\", \"r\");\n",
                                                    j, buffer);
}
fscanf (nodefile, "%d\n", &outfiles);
for (j=1; j<=outfiles; j++) {
    fscanf (nodefile, "%s\n", buffer);
    fprintf (process, "    outfile%d = fopen(\"%s\", \"w\");\n",
                                                    j, buffer);
}
fclose(nodefile);

fprintf (process,
        "\n    if ((fd = exopen(dev, nprocs, src)) < 0) {\n";
fprintf (process, "        printf (\"Failed to access %\");\n");
fprintf (process, "d nodes\", nprocs);\n");
fprintf (process, "        exit (1);\n    }\n");

for (i=0; i<nprocs; i++) {
    fprintf (process, "    src = %d;\n", i);
    fprintf (process, "    if (expload(fd, proc%d, src) < 0) {\n", i);
    fprintf (process, "        printf (\"Failed to load program %\");\n");
    fprintf (process, "s\", proc%d);\n", i);
    fprintf (process, "        exit (2);\n    }\n");
}

fprintf (process, "    src = 0;\n\n    printf (\"\\");
fprintf (process, "nEnter the number of iterations:\");\n");
fprintf (process, "    scanf (\"%\");\n");
fprintf (process,
        "ld\", &iterations);\n\n    exstart(fd, ALLNODES);\n");
fprintf (process, "    exmain (fd, ALLNODES);\n");
fprintf (process, "    exbroadcast (&iterations, mapping[0],
        sizeof(iterations), ALLNODES, NULLPTR, &types[%d]);\n", edges);

/* generate the iteration loop */
fprintf (process, "\n    for (i=0; i<iterations+%d; i++) {\n",
                                                overallmax);
fprintf (process, "        %s;\n    }\n\n", theNodes->nodename);

/* add if timing selected */
if (time) {
    fprintf (process, "    for (src=0; src<nprocs; src++) {\n");
    fprintf (process, "        exread (times, sizeof(times), &src,
                                                &types[%d]);\n", edges);
}

```

```

        fprintf (process, "          printf (\\"proc %");
        fprintf (process, "d : %");
        fprintf (process, "ld microseconds\\n\\n", src,
                                times[1]-times[0]);\\n    }\\n");
    }
    fprintf (process, "\\n");

    for (j=1; j<=infile; j++) {
        fprintf (process, "    fclose (infile%d);\\n", j);
    }
    for (j=1; j<=outfile; j++) {
        fprintf (process, "    fclose (outfile%d);\\n", j);
    }

    fprintf (process, "\\n    fclose (fd);\\n\\n    return 0;\\n");
}

```

F. RPS PROFILER MAIN PROGRAM

```

/* This is the main program for the RPS Profiler */

#include "ttest.h"

int
main()
{
    createProcesses();
    return 0;
}

```

G. PROFILER HEADER FILE

```

/* This file is ttest.h. It is the header file for the RPS Profiler
   functions */

#ifndef __TTEST_H
#define __TTEST_H

/* Creates the files for the code generation and calls all functions
   that generate code */

void createProcesses();

/* Generates the code for any processor based on the nodes assigned to
   it, The location of the other nodes, and the edges of the graph */

void makeProc (char filename[9], char funcname[20]);

/* Generates the code for the host PC processor based on the I/O routine
   provided and the location of nodes that require I/O */

```

```
void makeHost ();
```

```
#endif
```

H. PROFILER SOURCE FILE

```
/* This file is ttest.c. It is the source file for all RPS Profiler  
functions */
```

```
#define MAXPROCS 16
```

```
#include "ttest.h"
```

```
#include <stdio.h>
```

```
char *incSuff = ".inc";
```

```
char *toSuff = ".to";
```

```
char *fileSuff = ".fil";
```

```
char *codeSuff = ".tcs";
```

```
FILE *process;
```

```
void
```

```
createProcesses()
```

```
{
```

```
    char hostName[7] = "host.c";
```

```
    char procName[7] = "proc.c";
```

```
    char filename[9];
```

```
    char funcname[20];
```

```
    FILE *batfile;
```

```
    batfile = fopen("compile.bat", "w");
```

```
    fprintf (batfile,  
            "c:\\tc\\bin\\tcc -ml -c -Ic:\\parasoftware\\hostinc host.c\n");
```

```
    fprintf (batfile, "c:\\tc\\bin\\tcc -ml -ehost.exe host.obj  
                    c:\\parasoftware\\lib\\exp rtc.lib\n");
```

```
    fprintf (batfile, "c:\\parasoftware\\bin\\tcc -o proc proc.c\n");
```

```
    process = fopen(hostName, "w");
```

```
    makeHost ();
```

```
    fclose(process);
```

```
    printf("Enter the file name for the node being tested: ");
```

```
    scanf("%s", filename);
```

```
    printf("Enter the function name for the node being tested: ");
```

```
    scanf("%s", funcname);
```

```
    process = fopen(procName, "w");
```

```
    makeProc (filename, funcname);
```

```
    fclose(process);
```

```

}

void
makeProc (char filename[9], char funcname[20])
{
    int j, numincludes;
    FILE *nodefile;
    char buffer[81];
    char currentFile[13];

    fprintf (process, "#include \"express.h\"\n\n");

    /* other includes */

    strcpy (currentFile, filename);
    strcat (currentFile, incSuff);
    nodefile = fopen(currentFile, "r");
    fscanf (nodefile, "%d\n", &numincludes);
    for (j=0; j<numincludes; j++) {
        fscanf (nodefile, "%s\n", buffer);
        fprintf (process, "#include %s\n", buffer);
    }
    fclose(nodefile);
    fprintf (process, "\n");
    fprintf (process, "int mapping = 0;\n");
    fprintf (process, "int types = 0;\n\n");

    /* node functions */

    strcpy (currentFile, filename);
    strcat (currentFile, codeSuff);
    nodefile = fopen(currentFile, "r");
    while (fgets(buffer, 80, nodefile) != 0) {
        j = 0;
        while (buffer[j] == ' ') {
            j++;
        }
        if (buffer[j] == '+' && buffer[j+1] == '+' && buffer[j+2] == '+'){
        }
        else {
            if (buffer[j] == '-' && buffer[j+1] == '-' &&
                buffer[j+2] == '-') {
            }
            else {
                fprintf (process, "%s", buffer);
            }
        }
    }
    fprintf (process, "\n\n");
    fclose(nodefile);
}

```

```

fprintf (process, "int\nmain()\n{\n  int i, iterations=1000;\n");
fprintf (process, "    long timing[2];\n");
fprintf (process, "\n    mapping = HOST;\n");
fprintf (process, "    timing[0] = extime();\n\n");
fprintf (process, "    for (i=0; i<iterations; i++) {\n");
fprintf (process, "        %s;\n", funcname);
fprintf (process, "    }\n\n");
fprintf (process, "    timing[1] = extime();\n");
fprintf (process, "    "
        exwrite (timing, sizeof(timing), &mapping, &types);\n\n");
fprintf (process, "    return 0;\n}\n");
}

```

```

void
makeHost ()
{
    FILE *nodefile;

    fprintf (process, "#include \"express.h\"\n");
    fprintf (process, "#include <stdio.h>\n\n");
    fprintf (process, "char *dev = \"/dev/transputer\";\n");
    fprintf (process, "char *proc = \"proc\";\n\n");
    fprintf (process, "int    types = 0;\n\n");
    fprintf (process, "int\nmain()\n{\n");
    fprintf (process, "    long    times[2];\n");
    fprintf (process, "    int    nprocs = 1, src = 0, fd;\n\n");
    fprintf (process, "    if ((fd = exopen(dev, nprocs, src)) < 0) {\n");
    fprintf (process, "        printf (\"Failed to access %\");\n");
    fprintf (process, "    d nodes\", nprocs);\n");
    fprintf (process, "        exit (1);\n    }\n\n");
    fprintf (process, "    if (expload(fd, proc, src) < 0) {\n");
    fprintf (process, "        printf (\"Failed to load program %\");\n");
    fprintf (process, "    s\", proc);\n");
    fprintf (process, "        exit (2);\n    }\n\n");
    fprintf (process, "    exstart(fd, ALLNODES);\n");
    fprintf (process, "    exmain (fd, ALLNODES);\n\n");
    fprintf (process, "    "
        exread (times, sizeof(times), &src, &types);\n");
    fprintf (process, "    printf (\"Node computation time = %\");\n");
    fprintf (process, "    "
        "ld microseconds\\n\", (times[1]-times[0])/1000);\n\n");
    fprintf (process, "    exclose (fd);\n\n    return 0;\n}\n");
}

```


LIST OF REFERENCES

- [AKI 93] Akin, C., *Efficient Scheduling of Real-Time Compute-Intensive Periodic Graphs on Large Grain Data Flow Multiprocessors*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1993.
- [BEL 92] Bell, H. A., *A Compile-Time Approach For Chaining and Execution Control in the AN/UYS-2 Parallel Signal Processor*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1992.
- [BOK1 81] Bokhari, S. H., "On the Mapping Problem," *IEEE Transactions on Computers*, v. C-30, pp. 207-214, March 1981.
- [BOK2 81] Bokhari, S. H., "A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System," *IEEE Transactions on Software Engineering*, v. SE-7, pp. 583-589, November 1981.
- [DIX1 93] Dixit-Radiya, V. A., and Panda, D. K., *Mapping and Scheduling in Distributed-Memory Systems using Temporal Communication Graph Model*, Technical Research Report, Ohio State University, Columbus, Ohio, January 1993.
- [DIX2 93] Dixit-Radiya, V. A., and Panda, D. K., *Task Assignment with Link Contention on Distributed-Memory Systems*, Technical Research Report, Ohio State University, Columbus, Ohio, April 1993.
- [DUN 94] Dundar, C. A., *Improvement of Janus Using Pegasus 1-Meter Resolution Database with a Transputer Network*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.
- [ELR 90] El-Rewini, H., and Lewis, T. G., "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *Journal of Parallel and Distributed Computing*, v. 9, pp. 138-153, June 1990.
- [HAM 92] Hammond, S. W., *Mapping Unstructured Grid Computations to Massively Parallel Computers*, Doctoral Thesis, Rensselaer Polytechnic Institute, Troy, New York, February 1992.
- [HIC 95] Electronic Mail from Mr. Arthur Hicken, May 3, 1995, Parasoft Corporation, Monrovia, California.
- [HOA 93] Hoang, P. D., and Rabaey, J. M., "Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput," *IEEE Transactions on Signal Processing*, v. 41, pp. 2225-2235, June 1993.
- [INM 85] INMOS Limited, Bristol, U. K., *IMS B004 Evaluation Board User Manual*, 1985.

- [INM 86] INMOS Limited, Bristol, U. K., *IMS B003 Evaluation Board User Manual*, 1986.
- [INM 89] INMOS Limited, Bristol, U. K., *The Transputer Databook*, Second Edition, 1989.
- [KAS 94] Kasinger, C. D., *A Periodic Scheduling Heuristic for Mapping Iterative Task Graphs Onto Distributed Memory Multiprocessors*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1994.
- [KER 70] Kernighan, B. W., and Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, v. 49, pp. 291-307, February 1970.
- [LOV 88] Lo, V. M., "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Transactions on Computers*, v. 37, pp. 1384-1397, November 1988.
- [NEG 94] Negelspace, G. L., *Grain Size Management in Repetitive Task Graphs for Multiprocessor Computer Scheduling*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1994.
- [PAR 90] Parasoft Corporation, *EXPRESS C User's Guide Version 3.0*, 1990.
- [SHU 92] Shukla, S., Little, B., and Zaky, A., "A Compile-Time Technique for Controlling Real-time Execution of Task-level Data-flow Graphs," *Proceedings of the 1992 International Conference on Parallel Processing*, v. II, pp. 49-56, August 1992.

INITIAL DISTRIBUTION LIST

- | | | |
|----|---|---|
| 1. | Defense Technical Information Center
Cameron Station
Alexandria, VA 22304-6145 | 2 |
| 2. | Dudley Knox Library
Code 052
Naval Postgraduate School
Monterey, CA 93943-5101 | 2 |
| 3. | Chairman, Code CS
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943 | 2 |
| 4. | Dr. Amr Zaky, Code CS/ZA
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943 | 2 |
| 5. | Lt Charles Brian Koman
81 Atlas Rd.
Basking Ridge, NJ 07920 | 1 |
| 6. | Dr. D. K. Panda
CIS Department
The Ohio State University
Columbus, Ohio 43210 | 1 |
| 7. | Dr. Man-Tak Shing, Code CS/SH
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943 | 1 |

